RADC-TR-88-11, Vol VI (of eight) Part B
Interim Technical Report
June 1988

NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1986 Knowledge Base Maintenance Using Logic Programming Methodologies

Syracuse University

Kenneth W. Bowen

This effort was funded partially by the Laboratory Director's fund.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss AFB, NY 13441-5700

RADC-TR-88-11, Volume VI (of eight), Part B has been reviewed and is approved for publication.

APPROVED: [illegible]

JOHN CROWTER
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: [illegible]

JOHN A. RITZ
Directorate of Plans & Programs

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-11, Vol VI (of eight), Part B
6a. NAME OF PERFORMING ORGANIZATION Northeast Artificial Intelligence Consortium (NAIC) 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)
6c. ADDRESS (City, State, and ZIP Code) 409 Link Hall Syracuse University Syracuse NY 13244-1240	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0008
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 62702F (over) · PROJECT NO. 5581 · TASK NO 27 · WORK UNIT ACCESSION NO. 13

11. TITLE (Include Security Classification)
NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1986 Knowledge Base Maintenance Using Logic Programming Methodologies

12. PERSONAL AUTHOR(S)
Kenneth W. Bowen

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM Jan 86 TO Dec 86	June 1988	70

16. SUPPLEMENTARY NOTATION
This effort was funded partially by the Laboratory Directors' Fund.

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	
12	07	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial Intelligence, Prolog, Knowledge Base, Logic Programming, Languages, Feasibility, Knowledge Base Maintenance

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose is to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress that has been made in the second year of the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photo interpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.

The specific topic for this volume is the use of logic programming methodologies for knowledge base maintenance.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL John J. Crowter	22b. TELEPHONE (Include Area Code) (315) 330-2973 · 22c. OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

UNCLASSIFIED

Block 10 (Cont'd)

Program Element Number	Project Number	Task Number	Work Unit Number
62702F	4594	18	E2
61101F	LDFP	15	C4
61102F	2304	J5	01
33126F	2155	02	10

UNCLASSIFIED

NAIC
Northeast Artificial Intelligence Consortium
1986 Annual Report

Volume VI
Part B

Knowledge Base Maintenance Using Logic Programming Methodologies

Kenneth A. Bowen, PI
Staff: Hamid Bacha, Kevin Buettner, Ilyas Cicekli, Keith Hughes, Andy Turk

Logic Programming Research Group
School of Computer and Information Science
313 Link Hall, Syracuse University
Syracuse, NY 13210

Table of Contents

6-B.1 Executive Summary

The on-going work of this project is focused on the development of extensions of the logic programming language Prolog which are suitable for implementing complex artificial intelligence applications, including that of maintaining consistency and logical structure for large dynamic knowledge bases. For a programming language to be suitable for such applications, it must on the one hand be sufficiently expressive enough to promote high programmer productivity in building applications, while on the other hand, it must be possible to implement the language in such a way that applications programs run with suitably high efficiency. Moreover, the language must have a sound semantics — many AI applications are too serious to envisage entrusting their implementation to languages without a sound basis.

Typically, there is a trade-off between the first two requirements. Low-level languages such as assembler or C can produce very efficient code, but programmers have a great deal of difficulty in constructing complex AI applications with low-level languages. On the other hand, many extremely high-level languages have been designed which, in principle, could support very high programmer productivity, but precious few of them generate code which executes with sufficient efficiency for real-world applications.

This project is developing a quite high-level language which generates code which appears to be efficient enough to treat substantial real-world applications. The developing semantics is able to draw on mathematical logic. The language being constructed — metaProlog — is an extension of the AI programming language Prolog. The pattern of the research is an interplay between the three major concerns: extending the language's expressability, proving the implementation feasability of the extensions, and developing semantics to support the extensions.

The work during 1986 focused primarily on proving the feasability of efficient implementations of the extended metaProlog language which had been developed during the previous years. The focus was on the development of an abstract theory of compilation and execution of extended Horn clause languages, together with work towards the construction of a prototype implementation of the results of these studies.

The products of the year indicate that a highly expressive extension of Prolog is indeed efficiently implementable and will possess a valuable semantics. A complete prototype implementation was constructed during the coming year. During the next year, we will initiate a cycle of exploratory AI applications studies and semantics studies building on the present basis.

6-B.2 Introduction

6-B.2.1 Background and Goals

The goal of this project is the development of extensions of the logic programming language Prolog which are suitable for implementing complex artificial intelligence applications, including that of maintaining consistency and logical structure for large dynamic knowledge bases. Such languages must simultaneously satisfy the following criteria:

- They are suitable for the expression of the complex reasoning *about* knowledge necessary for the description, use, and maintenance of large knowledge bases in advanced intelligent systems.

- They possess sufficiently efficient implementations for realistic use in large-scale applications.

- They possess a coherent, logically sound declarative semantics.

The natural logical representation of the assertions of a knowledge base is to identify them with the facts and rules of the logic programming language. Such a collection of facts and rules is called a *theory*. To adequately model the dynamic character of the knowledge base, one must be able to efficiently manipulate alternative theories, a (possibly virtual) sequence of such alternative theories thereby providing a representation of the changing knowledge base. Moreover, it is desirable that the process of manipulating alternative theories be logically well-grounded. Present-day Prolog, while it has highly efficient implementations, does not meet these requirements. This is because pure Prolog only provides for one monolithic program database (theory) relative to which goals are solved (theorems are proved). Consequently, one cannot directly represent a changing KB. Practical Prolog adds the assert/retract facility to accomplish such a representation. This facility allows the program to destructively modify the program database on the fly. However, this has negative aspects:

- The logical basis of the system is destroyed;

- Since the KB is intertwined with the management code, this is very poor programming practice for large systems;

- It still does not allow one to express and maintain simultaneous alternative KBs or contexts.

The project is developing so-called *metalevel* extensions of Prolog meeting the following requirements:

- The extension allows one to express alternative and changing KBs;

- The extension has a logical basis;

- The implementation methodology for Prolog can be expanded to efficiently implement the metalevel extensions.

The metalevel extensions have the character that logical concepts which are implicit in Prolog systems are made explicit in the extension. In particular, theories, which are only implicit in Prolog, become explicit first-class objects capable of being the values of variables and of being dynamically constructed and modified in a logical manner. While the immediate motivation for this treatment of theories was the representation of change in knowledge bases, a pleasant side-effect is the ability to cleanly implement a number of classical artificial intelligence knowledge representation schemes such as frames and semantic nets. A similar approach is being explored for the problem of control, though it is not yet as well developed.

The key organizing principle we adopt in each aspect of the endeavor can be termed *meta-level abstraction*. As a scientific analytical technique it is a well-established technique. Its use in this setting has produced new viewpoints. In general, it directs us to render explicit conceptual entities which were previously implicit in the given endeavor. Specifically:

- At language level, it has led us to make databases, which we call *theories*, explicit objects (with surprisingly powerful consequences). It is leading us further to make other, more computationally-oriented objects, such as continuations, similarly explicit.

- At the semantic level, it leads to the incorporation of previously external semantic entities, such as evaluations and proof machinery, into the basic semantic structures (models).

• At the implementation level, it leads to the explicit design of a bridge abstract machine appropriately situated between language compiler and real hardware.

6-B.2.2 Approach of the Investigation

Endeavors at language extensions -- not restricted only to Prolog -- are surprising delicate and difficult to achieve. It is relatively easy to conceptually design extremely high-level langauges. One need only turn to one's favorite powerful mathematical tools to find inspiration for striking formal languages. If one follows mathematical (or broader scientific) paths, it is usually not overly difficult to also provide a meaningful semantics. But it is extraordinarily difficult to also provide efficient implementation methods for such languages, as the evidence of the last three decades of artificial intelligence and theorem-proving research amply demonstrates. Consequently, it is extremely important that the three aspects of the research effort proceed hand-in-hand, allowing none to outstrip the others. This leads to the following parallel intertwined activities:

• Experimental examination of proposed language constructs in intelligent systems applications.

• Examination of semantic renderings of the proposed constructs.

• Exploration of modifications to the abstract machine to efficiently support the new constructs.

The primary activities constituting this approach are suggested in Figure 6-B-1.

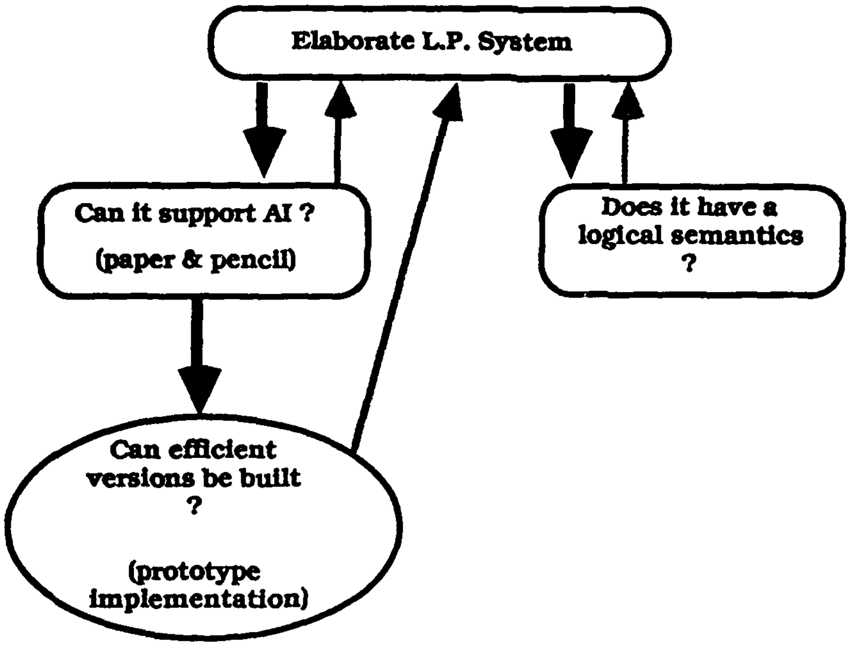
Elaborate L.P. System
Can it support AI ?
(paper & pencil)
Does it have a logical semantics ?
Can efficient versions be built ?
(prototype implementation)

Figure 6-B-1. Overall Research Approach.

The target products for the various activities are suggested in Figure 6-B.2.

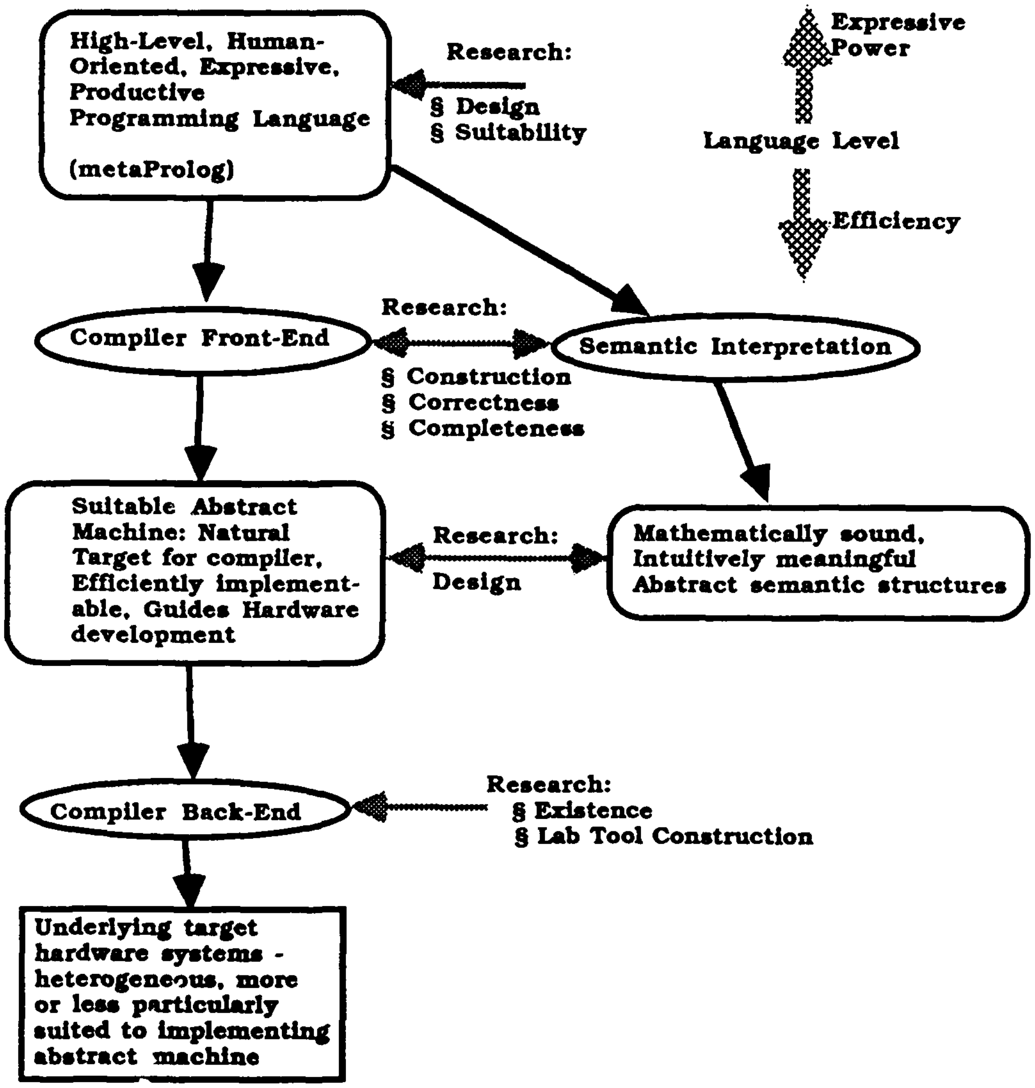
High-Level, Human-Oriented, Expressive, Productive Programming Language
(metaProlog)
Research:
§ Design
§ Suitability
Expressive Power
Language Level
Efficiency
Compiler Front-End
Research:
Semantic Interpretation
§ Construction
§ Correctness
§ Completeness
Suitable Abstract Machine: Natural Target for compiler, Efficiently implement-able, Guides Hardware development
Research:
Design
Mathematically sound, Intuitively meaningful Abstract semantic structures
Compiler Back-End
Research:
§ Existence
§ Lab Tool Construction
Underlying target hardware systems - heterogeneous, more or less particularly suited to implementing abstract machine

Figure 6-B.2. Target Products of the Research Activities.

It is important to stress the research aspects of the elaboration of the abstract machines and the construction of the associated compilers:

• There is a definite conceptual interplay between the top-level language design and the elaboration of the abstract machine, particularly in the present second-stage, that of making certain computational constructs linguistically explicit. The language versions of computational constructs (e.g., continuations) must be sufficiently abstract to be powerful and productive tools for applications programmers dealing with intelligent applications. The appropriate abstract level appears to be that of major architectural aspects of the abstract machine. Consequently, the precise structure of the abstract machine which it is possible to realize in turn conditions the detailed form of the high-level language extension.

• The abstract machine must remain situated at a level which makes it possible to provide efficient execution of the high-level language. Thus, the gap between abstract machine and high-level language must be such that correct, complete, and efficient compilers from the high-level language to the abstract machine can be constructed with tolerable effort, while on the other hand, the gap between the abstract machine and the (various) underlying real hardware must remain small enough to allow highly efficient real implementations. {In our laboratory prototypes, the compiler really outputs abstract machine instructions, and the abstract machine itself is emulated by byte-code interpreters running on SUNs and VAXs. In high-performance implementations, the abstract machine would fade into the background underlying native code compilers -- but would still have an explicit existence in the high-level language constructs.}

The construction of laboratory prototypes is essential. Beyond the obvious test of raw efficiency, they are necessary for true testing of the expressive adequacy of the high-level language extensions. The relatively small demonstration programs thus far constructed argue strongly for this expressive adequacy. However, the true test will lie in the construction of real applications-scale programs. Without tolerably efficient test systems, such applications tests will not be possible.

In the evaluation of these efforts, it is worth knowing the level of efficiency for which we are striving. For this, we will use the admittedly crude, but useful, naive reverse benchmark: the number of logical inferences (rule applications) per second (LIPS) required to reverse lists of lengths between 100 and 1000 elements according to the following naive algorithm (where [H | T] is the list with head H and tail T):

i) The reverse of the empty list [] is just the empty list [].
ii) The result of reversing [H | T] is the result of appending the list [H] (whose only element is H) to the result of reversing the tail list T.

On a VAX 780, our prototype runs at approximately 9,000 LIPS, a little less than 1/2 the speed of the premier commercial Prolog system supplied by Quintus.

Both systems utilize roughly the same abstract machine conception, with ours implemented by a byte-code interpreter, while Quintus's is implemented via threaded code. Recent work by several groups in the U.S and Europe has shown that native code compilers for Prolog can execute at 200,000 LIPS on SUN 3/280 systems. Scaling this by a factor of two for similar systems utilizing the Motorola 68030 chip yields speeds approaching 400,000 LIPS. Applying another scale factor of 2 or more for systems utilizing anticipated follow-ons to the 68030 generation chip, one begins approaching or exceeding one million LIPS for basic Prolog systems. Applications experience indicates that such speeds will make some very serious intelligent systems possible. Our experience in evolving the meta-level system prototypes from underlying Prolog systems indicates that one will be able to achieve similar speeds with compiled metaProlog code.

One final note concerning the abstract machines: They are more than useful compiler fictions. They are appropriate starting points for the design of real hardware. One of the successful abstract Prolog machine designs was elaborated by Warren and Tick into a pipelined complex instruction set (CISP) hardware design which has been constructed by Despain at Berkeley and is being commerically offerred by Xenologic, Inc. The same abstract machine provided the starting point for the very promising RISC designs being produced by Mills at Arizona State University. Recent evidence suggests that functional languages (LISP, Scheme) can also be given extremely efficient implementations over such abstract machines in a manner providing for close integration of the logic and functional languages. This raises the very real possibility of integrated hardware support for both approaches to artificial intelligence programming. Our initial designs for abstract machines supporting the second stage of our meta-level Prolog extensions provide support for both the types of continuations needed by the logic languages (metaProlog) and by advanced functional languages such as Scheme.

The discussion thus far has focused on language expressiveness and processing speed. However, great expressiveness and processing speed will be useless without a corresponding ability to access and manipulate the large amounts of data characteristic of many intelligent applications. We have been greatly concerned with this problem. Our goal is to provide interface mechanisms between the Prolog-based systems and external relational database management systems such as INGRES or ORACLE, or new systems possibly with direct hardward support. Such interfaces must go beyond simply allowing the Prolog-based system to view the tabular relational data as virtual Prolog facts, but must provide for integration of complex entity-relation architectures on the Prolog side with the flat tables on the DBMS side. It must also provide, if possible, for integration of the indexing methods utilized by both the DBMS and Prolog-based systems for efficiency. We will begin exploring the construction experimental interfaces between our basic Prolog prototype system and INGRES to study this issue. As a result, we have provided deep facilities in the present abstract machine design to support such interfaces. These will of course propagate into the more advanced designs.

In sum, our argument is that all aspects of our effort constitute basic research into the development of viable high-level programming systems for large-scale intelligent systems. No single aspect can be successfully carried to completion without the others.

6-B.3 The Prolog Compiler Platform

Since the metaProlog extension of Prolog is intended to build both on the expressive character of the Prolog language as well as its implementation technology, it was important that we have available a high-quality Prolog compiler in source code form which we would be free to modify and to distribute the results of those modifications. Previous work (under this grant and others) had explored the question of writing interpreters for the metaProlog extensions in both C and LISP (the latter on an LMI LISP machine). These experiences, together with the observed moderate efficiencies of Prolog interpreters, led us to the conclusion that it would be impossible to construct realistic test applications of the metaProlog extensions if we adopted an interpreter-based approach. Thus, we felt it necessary to pursue a compilation-oriented path. This is particularly true since the young state of logic-based language compilation technology made many questions of just how to compile advanced logical constructs a research problem rather than a straight-forward compiler-engineering problem as would be the case for more conventional languages.

In 1984-85, we we approached this problem, the only known non-commercial Prolog compiler was that built by D.H.D. Warren and his associates at the University of Edinburgh (Warren and Pereira [1977], Pereira et al. [1978]). While we did have informal access to the source code for this system, there were three fundamental reasons which prevented us from building on it directly:

- It was based on a particular logic programming implementation technology — *structure-sharing* (Warren [1977]) — which is less suitable than a newer technology — *structure-copying* (Mellish [1982]);

- It was very closely tied to the architecture of the DEC-10 computer and not highly portable to more modern workstations;

- It legally belonged to the British Science Research council and we would not be free to distribute modifications which we might make to it.

Consequently, we had to start fresh and construct our own basic Prolog compiler which we could then evolve into a compiler for the metaProlog extensions. We did this with considerable success, reporting the results in a collection of papers presented at the Third International Conference on Logic Programming held in London in the summer of 1986 (Bowen et al. [1986], Buettner [1986], and Turk [1986] — cf. Appendixes 6-B-A, 6-B-B, and 6-B-C). Not only was the system — based on a byte-code interpreter for the underlying Abstract Prolog Machine — remarkably efficient, but we evolved some striking new compilation techniques (Buettner [1986] and Turk [1986] — cf. Appendixes 6-B-B and 6-B-C). Due to these techniques, the resulting system is not only highly efficient, but is an incremental and interactive compiler for the complete Prolog language supporting garbage collection. No other system, including commerical systems, possesses any of these properties except efficiency. The entire system was written in the C language under UNIX, so that in consequence it is highly portable to a wide range of modern workstations and mainframe computers.

6-B.4 Extension to the metaProlog Compiler(s)

6-B.4.1 Outlines of metaProlog

The fundamental operator of standard Prolog systems is a one-place operator (usually written **call(...)**) which invokes a search for a deduction of its argument from the implicit program database parameter. The heart of the metaProlog extension is the implementation of the full logical deducibily relation instead of the restricted version embodied in 'call'. The system contains metavariables which not only range over formulas and terms, but also range over sets of formulas (*theories*). The fundamental operator of the system is a two-place operator, usually written **demo(___ , ...)**, which invokes a search for a proof of the goal formula appearing as its second argument from the theory (or program) appearing as its first argument.

In the metaProlog system, the only analogue of the standard Prolog database is the global database containing system built-in predicates. All other databases are the values of Prolog variables and are set up either by reading them in from files or by dynamically constructing them using system predicates. Besides the system predicate **demo(___,...)**, the system predicates include **addTo(___ , ___ , ___)** and **dropFrom(___ , ___ , ___)** which build new theories from old ones by adding or deleting formulas. Thus for example, one might find the body of a clause containing calls of the form

..., addTo(T1, A, T2), demo(T2, D),...

where the theory which is the value of T1 has been constructed by the earlier calls. The effect of this goal would then be to construct (in an efficient manner) a new theory T2 resulting from T1 by the addition of the formula A as a new axiom; then the system invokes a search for a proof of the formula D from the theory T2. Since demo implements the proof relation, such programs as this preserve the logical semantics of Prolog while providing for the dynamic construction of new databases from old.

The correctness and completeness of an implementation of 'demo' are expressed by *linking rules*:

If demo(T, A), then A is derivable from T.

If A is derivable from T, then demo(T, A).

These rules provide the justification for the implementation of calls on demo in the abstract metaProlog machine as context switches. In essence, at most times the machine behaves as a standard Prolog machine with the current theory (the analogue of the usual fixed program database) indicated by a register. When a call demo(T, A) is encountered, the database (theory) register is changed to point to T and a new search for a deduction of A is begun. Thus the efficiency of standard Prolog computations is preserved and the overhead of meta-level computation is localized in the construction of new theories from old together with quite fast context switches. This method of reflection has been utilized heavily in constructing the abstract metaProlog machine as a conceptual extension of a primitive storage management machine.

The metaProlog system is designed to be syntactically compatible with the Edinburgh system, to preserve the standard logical semantics, and to incorporate the full two-place proof relation. Thus, while syntactically quite similar to Edinburgh Prolog, metaProlog provides a quite different set of built-in meta-level predicates and allows the metavariables greater range than the Edinburgh system.

There is one major syntactic difference between the two systems. For reasons which were detailed in Bowen and Weinberg [1985], we require that the implicit universal quantifiers on clauses be made explicit. Thus, for example, the Edinburgh clause

```
append([Head | Tail], RightSeg, [Head | Result_Tail]) :-
    append(Tail, RightSeg, Result_Tail).
```

would be written

```
all [head, tail, rightSeg, result_Tail] :
    append([head | tail], rightSeg, [head | result_Tail]) :-
    append(tail, rightSeg, result_Tail).
```

If the clause contains only one variable, the list brackets in the quantifier can be dropped. Additionally, we allow the programmer to optionally use <— instead of :-, and to connect the literals in the body of a clause by & instead of comma. Thus the Edinburgh clause p(X) :- q(X),r(X).

might be written

all x : p(x) <— q(x) & r(x).

An expression which contains no meta-variables (but may have object-level variables occurring listed in the quantifer) is called a *closed formula*. A *theory* is either 'he *empty theory* (designated by **empty_theory**) or is of the form

A & U

where A is a formula and U is a theory. A theory is a *definite theory* if all of its formulas are closed. (Note that quantified variables may be present in definite theories.) Theories contained freely occurring logical variables are called *indefinite theories*. [Whether or not the programmer is allowed to directly write free logical variables is a matter of design controversy. But indefinite theories can always be created by metaProlog programs at run-time.]

The proposed built-in predicates of metaProlog include most of those of Edinburgh Prolog with the exception that all those concerned with the "program database" are excluded. Instead, a two-place demo, a three-place demo, the two three-place predicates addTo and dropFrom, the binary predicate axiomOf, and the unary predicate current are to be built-in predicates of metaProlog.

demo(Theory, Goal)

This call invokes a subsidiary (Prolog) computation which attempts to derive **Goal** on the basis of the program **Theory**. If **Theory** or **Goal** are are not fully instantiated, meta-variables occurring in either may be bound if a successful computation can be found.

Calls on demo support a convenient idiom for describing implicit unions of theories. Specifically, a call of the form

demo(Theory1 & Theory2, Goal)

is logically equivalent to the call

demo(Theory3, Goal)

where Theory3 is the ordered union of Theory1 and Theory2 in the following sense: If theories are regarded as the ordered list of their axioms, then Theory3 satisfies

append(Theory1, Theory2, Theory3).

However, the system does not physically create Theory3, but regards the expression Theory1 & Theory2 as a description of a *virtual theory*. In effect, when searching for a rule or fact to apply to a selected subproblem of the current goal, it first searches Theory1 for a candidate, and only on failing to find such a candidate in Theory1, it then searches Theory2. Another usage supported is the explicit indication of the axioms of the theory. Namely, if it is desired to search for a deduction of G from A1,...,An, this is achieved by the call

demo([A1,...,An], G).

(Internally, theories are represented in several forms. The simplest is just that of a list of axioms, with no special indexing. Retrieval from such a theory amounts to a linear search of the list. Thus, for all but relatively small theories, computation from such a theory will be intolerably slow.)

The two usages can be combined, as in the calls:

demo([A1,...An] & Theory2, Goal)

demo(Theory1 & [A1,...An], Goal).

addTo(Theory, Clause, NewTheory)

If **Theory** is bound to a theory and **Clause** is a (closed or partially instantiated) formula and **NewTheory** is an uninstantiated variable, this call causes **NewTheory** to be bound to a theory obtained from Theory by adding Clause as a new axiom. What actually happens is that the orignal theory bound to **Theory** is physically modified by the addition of **Clause**, providing fast access to the **NewTheory**, with relatively slower access to the clauses of the old **Theory**. The variable **Theory** is rebound to an internal representation of the result of dropping **Clause** from the theory now bound to **NewTheory**, in a manner inverse the the common method of representing arrays in logic. Thus the original theory bound to **Theory** is still logically available via **Theory**, but access to it is a bit slower. If **Theory** is not bound to a theory or if **NewTheory** is instantiated a run-time error occurs. Note that, unlike the treatment of assert in conventional Prolog, metavariables occurring in **Clause** are NOT converted to universally quantified object variables in the assert fact or rule.

dropFrom(Theory, Clause, NewTheory)

Under the restrictions of addTo, this call causes **NewTheory** to be bound to the theory resulting from the deletion from **Theory** of the first occurrence of an axiom of **Theory** which matches **Clause**. The internal representations and run-time errors are similar to those for addTo.

axiomOf(Theory, Clause)

This call succeeds if **Clause** has been recorded as an axiom of **Theory**. If **Clause** is uninstantiated, it will be bound to the first axiom of **Theory**. Backtracking into this call will cause **Clause** to be successively bound to the axioms of **Theory**. **Theory** must be bound or a run-time error occurs.

current(Theory)

This call is equivalent to the Prolog definition

demo(Theory, current(X)) :- X = Theory.

Note that axioms of the current theory can be directly accessed via the goal

```
<—current(Theory), axiomOf(Theory, Clause).
```

6-B.4.2 metaProlog Programming Example #1: The Inland Spills Expert

Here we discuss an adaptation of a program to manage inland chemical spills at the Oak Ridge National Laboratory. The problem is discussed in detail in Hayes-Roth, et al.[1983]. Our metaProlog program for this problem was strongly influenced by the Rosie SPILLS program (Fain et al.[1982]), from which the following problem statement was drawn:

The SPILLS program locates and identifies hazardous chemical spills, given a database describing the location of the spill, the location of chemical storage containers, and a description of the drainage network. SPILLS evolved as an answer to a problem posed at the expert Systems Workshop in San Diego, August 1980 ...The problem involved the creation of an *on-line assistant to* aid a crisis control team in the location and containment of chemical spills at the Oak Ridge National Laboratory. Two experts in the field plus a preliminary report ... provided the necessary expertise.

The Oak Ridge Laboratory has approximately 200 buildings scattered over a 200-square-mile area, many storing hazardous chemicals in *containers ranging* in size from small 1-gallon bottles or cans to huge 5,000-gallon storage tanks. The drainage network under the building collects all spills and discharges them into White Oak Creek, a waterway running along one side of the lab's complex. When chemical discharges are noticed in the creek they must be traced back to some source (a storage container in or near a laboratory building) so the leak can be stopped and the spill contained.

The SPILLS program attempts to locate the source of the spill by tracing the flow of spilled material through the drainage basin back to the source. This search method requires a human assistant who must go out in the field and actually look into the drainage networks at various check points (usually manholes) to see if the spill material is there. There are so many manholes (hundreds) that it is not practical to check them all for traces of the spill. Instead, the program uses the information at its disposal to decide which checkpoint would provide the maximum amount of information at any given time, and recommends that the assistant examine that checkpoint. After the program is told the result of that examination, it recalculates the new optimal checkpoint, and the process continues until the source is found.

Besides processing reports on the location of the spill material, the program processes reports describing the characteristics of the spill material. It attempts to determine what the material is and how much of it has been spilled. This information, in turn, helps reduce the number of possible locations to be checked.

One of the many drainage basins at Oak Ridge is shown in Figure 6-B.3.

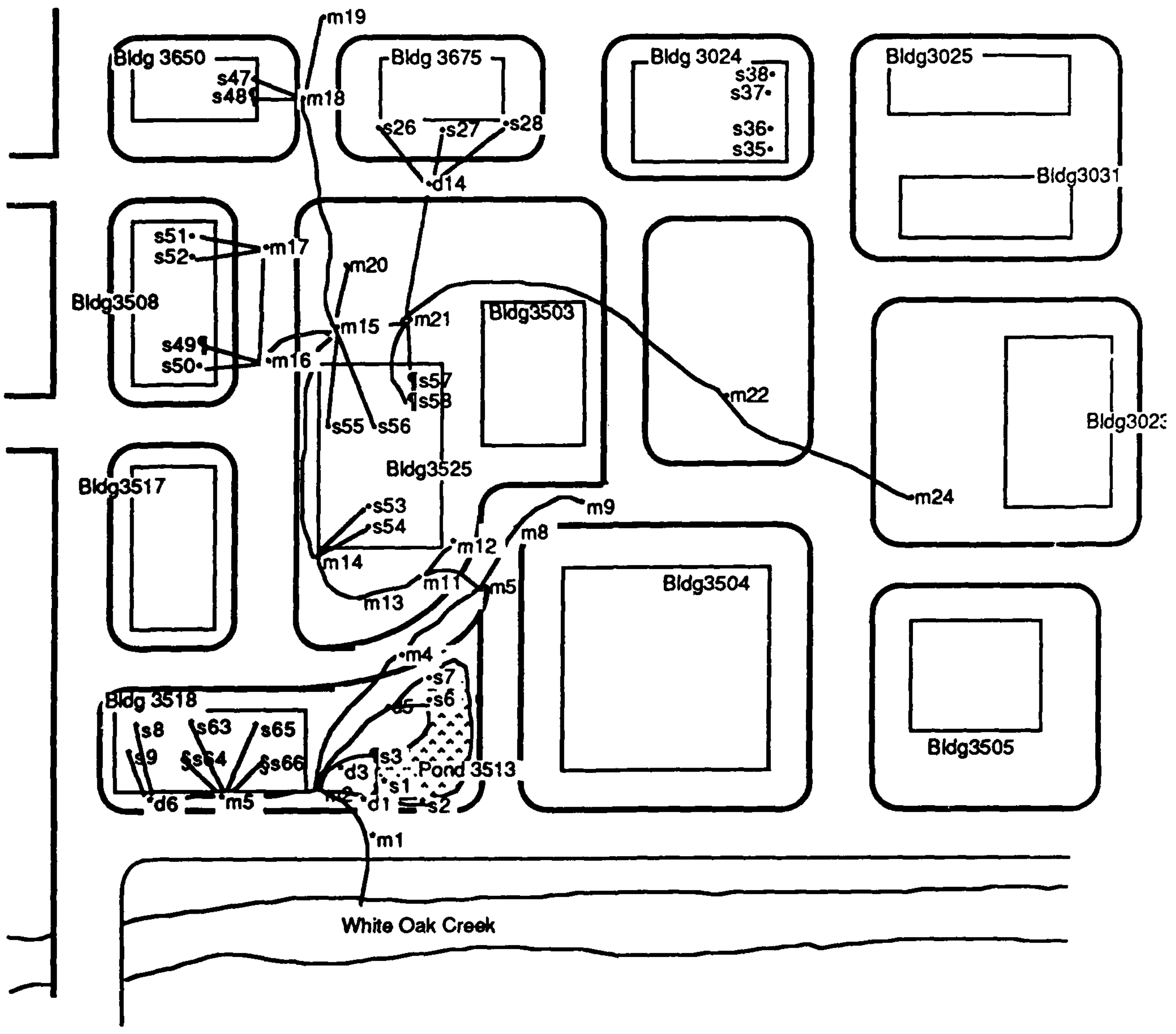

Figure 6-B.3.
Part of One of the Drainage Basins at Oak Ridge National Laboratory.

It is evident that the basin forms a tree with its root at the White Oak Creek effluence and its tips consist of the various sources. The problem then is one of exploring this search tree starting at the root (where the spill is first observed) and locating the offending tip (a leaky tank). The difficulty in this exploration (which involves a human assistant going out and looking into the manholes) is the large size of the tree. Thus the program, just as a human, will attempt to apply knowledge to minimize the portion of the tree which must be examined.

Conceptually, at the outset of the search all of the tips are possible sources of the spill. As knowledge (of the nature of the material and of the manholes at which it has been observed) is accumulated, various of the tips are eliminated as possible sources. Our metaProlog program, named OakRidge, emulates this approach by maintaining a dynamic theory representing the current state of its knowledge. At the outset, it consists solely of a collection of assertions to the effect that each of the sources in the basin is a possible source of the spill:

```
    possibleSource(s(1)).
    possibleSource(s(2))
    . . . .
    possibleSource(s(70)).
```

As the search progresses and various sources are eliminated, the appropriate possibleSource(s(N)) assertions are deleted, while assertions regarding the nature and properties of the spill material are added along with assertions about the manholes at which it has been observed.

The knowledge which the program possesses at the outset is broken up *into* various static theories which are utilized by the reasoning processes. The knowledge of the topology of the network, the nature of its nodes, and the nature of each of the sources is contained in a theory called srcs:

```
isPond(pond(3513)).
outfall(woc(6)).
isBuilding(building(3023))
. . .
isBuilding(building(3550)).

all N :  isDrain(drain(N)) <— between(0, N, 16).
all N :  isManhole(m(N)) <— between(0, N, 47).
all N :  isSource(s(N)) <— between(0, N, 71).
all N :  near(s(N), pond(3513)) <— between(0, N, 8).
all N :  in(building(3023), s(N)) <— between(43, N, 46)
. . .
all N :  in(building(3504), s(N)) <— between(10, N, 17).
all N :  in(building(3504), s(N)) <— between(67, N, 70)
. . .
parent(woc(6), m(1)).
parent(m(1), m(2)).
parent(m(2), m(3)).
parent(m(2), m(4)).
. . .
all N : parent(m(5), drain(N)) <— between(7, N, 10).
. . .
all [M, N] :  parent(drain(N), s(M)) <— between(2, N, 5) & M is N+1.
. . .
contains(s(1), gallons(2000), [transformer, oil]).
contains(s(2), gallons(1000), [gasoline]).
contains(s(3), gallons(10), [acetic, acid]).
    . . .
```

Other static theories contain knowledge about how to infer the nature of the spill material from its properties, how to infer the next manhole to examine, and how to eliminate possible sources. These will be described below. The top level of the program appears as follows:

```
oakRidge <— investigate(null).

all [currentData, updatedData, d0, d1, d2, d3] :
   investigate(currentData) <—
      write('Report please:') & nl  &
      getReport(currentData,updatedData)  &
      workOnMaterialType(updatedData,d0)  &
      workOnMaterial(d0,d1)  &
      workOnMaterialVolume(d1,d2)  &
      workOnMaterialSource(d2,d3) & !  &
      dispatch_investigate(d3).

all updatedData :
   dispatch_investigate(updatedData) <—
      finished(updatedData) & !.

all data :  dispatch_investigate(data) <—
   ! &  investigate(data).
```

The predicate getReport is the "natural language" front-end which obtains information to the user. The details of its definition are included in the appendix to this section. The variable CurrentData is bound to a theory which represents the current information regarding the spill being investigated. This is extended by getReport with the information obtained. The workOn_______ predicates are concerned with inferring the general nature, specific identitiy, volume of, and source of, the spill. They each take the current information as input, and add to it any inferences they may make to produce their output. Finally, dispatch_investigate determines whether or not the source has been determined, and hence, whether or not the investigation should be (tail recursively) continued.

Consider the predicate workOnMaterialType. It is concerned with the problem of inferring the general nature of the spill material. It bases its deductions on the current information regarding the spill material (the variable UpdatedData) and a small theory "typeOfMaterial" which encodes the "expertise" for inferring the types of spill materials in this setting. The definition of workOnMaterialType runs as follows:

```
all [data, extendedData, x ] :
   workOnMaterialType(data, extendedData) <—
      write('Trying to determine material type...') & nl  &
      demo(typeOfMaterial & Data,type_of_material(spill,x)) &
      ! &
      addTo(data, type_of_material(spill,x), extendedData)   &
      write('The type of the spill material is ')  &
      print(x) & nl & nl.

all data :
   workOnMaterialType(data, data) <—
      write('Can''t determine the type of the spill material now...')  &
      nl & nl.
```

The theory typeOfMaterial contains the following axioms:

```
all n :   type_of_material(spill, [oil]) <—
   appears(spill_solubility, [low])  &
   approximates(ph_of_spill, [n])  & 5 < n &  n < 9.
```

```
all n :  type_of_material(spill,[base]) <—
    appears(spill_solubility,[high])  &
    approximates(ph_of_spill,[n])  & 8 < n.

all n :  type_of_material(spill,[acid]) <—
    appears(spill_solubility,[high])  &
    approximates(ph_of_spill,[n])  & n < 6.
```

The predicate workOnMaterial attempts to infer the specific composition of the spill. It bases its work on the current information at the time of its invocation (the theory d0) together with the theory "materialType" contains the expertise for inferring the spill composition. The definition of workOnMaterial runs as follows:

```
all [data, extendedData, x] :
    workOnMaterial(data, extendedData) <—
        write('Trying to determine material...') & nl  &
        demo(materialType & data, consists(spill,of(x))) & !  &
        addTo(data, consists(spill,of(x)), extendedData)  &
        write('The spill consists of ')  & write(x) & nl & nl.

all data :  workOnMaterial(data,data) <—
    write('Can''t determine what the material is now...')    & nl & nl.
```

The theory materialType contains the following axioms:

```
consists(spill,of([sulpheric,acid])) <—  iss(sulphate_ion_test,[positive]).

consists(spill,of([gasoline])) <—
    type_of_material(spill,[oil]) & smells(spill,of([gasoline])).

consists(spill,of([diesel,oil])) <—
    type_of_material(spill,[oil]) & smells(spill,of([diesel,oil])).

consists(spill,of([acetic,acid])) <—
    type_of_material(spill,[acid]) & smells(spill,of([vinegar])).
```

```
consists(spill,of([hydrochloric,acid])) <—
    type_of_material(spill,[acid]) &
    has(spill,[pungent,'/',choking,odor]).
```

While the content of the rules in these theories is not particularly deep, nonetheless, they exhibit the necessary characteristic of expert system rules: the clear expression of whatever expertise they embody.

The remainder of the example has a similar character.

6-B.4.3 metaProlog Programming Example #2: Frames

Frames (cf. Minsky [1975] or Kuipers [1975]) are among the more widely-used and powerful techniques in artificial intelligence programming. Abstractly, frames a just a collection of "slots" with labels. Though they have many manifestations, there appear to be two crucial properties common to most implementations:

i) The collection of slots making up an individual frame are physically grouped together in storage guaranteeing that they can be accessed as a group; if the frame is a first-class object, they can move about together.

ii) Besides being fillable with rather ordinary entities (atoms, numbers, compound expressions), (certain) slots can be filled with references to other frames. These references can be used to organize collections of frames in various kinds of hierarchies which can be exploited by systems which are utilizing the collection.

Typically, a frame carries an identifier which specifies it uniquely in the collection. Thus, a frame labelled "elephant" might contain the following slots among others:

```
a_kind_of : mammal
color : grey
number_of_toes : 4
```

Another frame, labelled "clyde" and intended to represent a particular elephant, might contain the following slots among others:

```
    a_kind_of : elephant
    home : london_zoo
```

The entry "elephant" in the frame for clyde might indeed be the identifier "elephant", or might be an internal direct reference to the frame representing generic elephants. In the former case, a frame processing system which was looking in clyde's frame and needed to obtain some information from the elephant frame (which is generic information generally true of all elephants) would first need to look up the identifier "elephant" in some internal table giving it the location of the generic elephant frame. Also, generic information is in such settings usually used as a default, and can be over-ridden by information in the specific frame. Thus if clyde were an albino, the frame for clyde could contain the slot

color : white

which would be accessed when clyde's color was needed. When attempting to obtain information from a particular slot in a particular frame, processing systems typically default to higher frames in the "a_kind_of" hierarchy only when the given slot is not present in the particular frame.

From a logical point of view (cf. Hayes [1977]), the slots of a frame and their contents can be viewed as assertions. Thus the frame for elephants could be taken to be equivalent to the collection

```
a_kind_of(elephant, mammal)
color(elephant, grey)
number_of_toes(elephant, 4)
```

while the frame for (the albino) clyde could be taken as being equivalent to the collection

```
a_kind_of(clyde, elephant)
home(clyde, london_zoo)
color(clyde, white).
```

But in metaProlog, the theory construct is designed specifically for the representation and manipulation of collections of assertions. Moreover, the collection of assertions making up a theory can indeed be physically grouped together, or else the actual arrangement is such that the access effects are very much as though the assertions were grouped together. Thus it is obviously natural in metaProlog to represent frames as (possibly small or large) theories containing assertions which correspond to the slots and their contents.

Given this point of view, it remains to be seen how we may organize the manipulation of these theories representing frames so as to achieve the effects produced by typical frame manipulation systems. (We will do this first from a straight-forward naive point of view; later we will refine the approach to achieve more compact storage utilization and the effects of direct embedded pointers from "a_kind_of" slots to other frames.)

First off, let us assume that we have a master theory called "animals" in which our work will be carried out. Recall that any theory can contain assertions about other theories since the latter are first-class objects. Within the theory "animals" we will find some assertions identifying frames:

```
is_frame(elephants, Teleph)
is_frame(clyde, Tclyde).
```

These assertions can be collectively viewed as a table associating the individual collections of assertions corresponding to a frame with the identifier naming the frame. (The actual implementation would allow access to the individual collections, given the frame identifier, at the low cost of sequential probes in two hash tables; in the revised version below, the cost is one probe in a single hash table, and then the following of a direct pointer.) The individual collections Teleph and Tclyde might have been initialized from files by calls such as

```
consult(Teleph, 'elephants.mysys').
```

New slot entries (i.e., new assertions) can be entered in the individual collection by use of the addTo built-in predicate. The question arises as to what transpires when one wishes to modify (update) an already existing slot value (i.e., in this setting remove an existing assertion and replace it by a new one). As will be discussed in Section 14, the implementation of theories is as a kind of "mutable array" which supports backtracking, yet provides all the speed of normal array access. In short, in a call

```
addTo(T1, A, T2),
```

the mutable array representing the theory to which the variable T1 is bound is actually updated by the insertion of A, this updated array is bound to T2, a descriptor referencing this updated array and A is bound to T1, and everything appropriate is trailed. The descriptor to which T1 is bound describes the original value of T1 in terms of A and the updated value currently bound to T2.

For our purposes here, it suffices to say that the theory representing the frame can be updated in such a way as to provide the same fast access as the original frame and yet still preserve the logical characater of the computation.

It remains to be seen just how the remaining actions of frame processors are effected in this context. This is most easily seen by considering the clauses which might be added to an ordinary Prolog definition of the metaProlog interpreter (i.e., the predicate demo). Briefly, one needs to add something like the following clauses to the set of clauses defining demo:

```
demo(Theory, Goal) :-
   Goal =.. [Predicate, Arg1 | RestArgs],
   demo(Theory, is_frame(Arg1, Frame_Theory)),
   demo(Frame_Theory, Goal).

demo(Theory, Goal) :-
   Goal =.. [Predicate, Arg1 | RestArgs],
   demo(Theory, is_frame(Arg1, Frame_Theory)),
   demo(Frame_Theory, a_kind_of(Arg1, B)),
   H =.. [Predicate, B | RestArgs],
   demo(Theory, is_frame(B, B_Frame_Theory)),
      demo(B_Frame_Theory, H).
```

As indicated above, we will later revised this to compact code and storage and increase efficiency. But for now, let us see just how this will work. Given the call

```
   demo(animals, color(clyde, X))
```

the first of the two clauses above will apply, so that the theory Tclyde will be retrieved by the second call in the body, and the third call will have the effect of binding X to white. On the other hand, given the call

```
   demo(animals, number_of_toes(clyde, N))
```

the first call will ultimately fail, causing the second clause to be invoked. After retrieving the theory Tclyde and succeeding in deriving that a_kind_of(clyde, elephant) from it, the theory Teleph will be retrieved and the call

```
demo(Teleph, number_of_toes(elephant, N))
```

will be run, finally binding N to 4, in the accepted manner of inheritance.

As naively described above, the organization of collections of frames is based on "master hash tables" containing pointers to the various frames. Given a modern workstation processor of sufficient power, this might not be a bad approach. However, the elements of the metaProlog approach provide a more sophisticated organization. The cost of the naive approach is encountered when following inheritance pointers such as "is_a_kind_of". The tracing of these pointers would involve multiple probes in the master hash table. However, these can be replaced by the tracing of pointer chains as follows.

The notion of a metaProlog name is similar to the natural language notion of name. It is a syntactic item which somehow directly refers to the object it names. The details of an implementation method are irrelevant so long as the name relation posseses the required abstract properties. Consequently, we are free to implement the name relation and names any way which provides the essential "referring" property of names. Since all the items referred to by metaProlog names are themselves syntactic items, the things named are ultimately just computer data structures which must reside at locations in memory. Consequently, a prime candidate for the implementation of names is the use of internal memory pointers referring to the locations of the data structures. Most likely these references will be more complex than raw pointers — for example, they might be tagged pointers. But metaProlog names are just metaProlog entities, no different in general character from other metaProlog entities, and consequently, names can participate in assertions just like all other entities. Hence, the inheritance assertions contained in a frame can refer to the super-ordinate frame using such a name of the super-ordinate frame. But then, following inheritance frames simply involves extracting the memory address from the name, and following the resulting pointer, etc.

6-B.4.4 The Key Simulator

The following is a description of the key metaProlog simulator, which was implemented using our basic Prolog compiler. The development of the ideas underlying this simulator was a key step in organizing the approach to actual compilation of metaProlog. This simulator was a preliminary test of the concept before embarking on the redesign of the Abstract Prolog Machine to support the necessary mechanisms The simulator is different from the previous metaProlog simulators in that clauses are now fully compiled, rather than interpreted. This compilation is achieved through the *guard literal*, an idea proposed by Andy Turk for compiling meta at the machine level.

The basic idea is that theories are denoted by lists. (In the actual metaProlog compiler, we replaced the use of lists at the Prolog level by the use of indexing instructions in the Abstract Prolog Machine.) When an addTo or dropFrom is done, a unique theory ID is generated. The path to this new theory is then the new theory ID appended to the end of the old theory descriptor. In the case of an addTo, a new clause is asserted, while in the case of a dropFrom, the old clause in the database is modified. For example, if the fact p is added to the empty theory ([]), a new theory descriptor of [0] would be created (addTo([],p,[0])). A rewritten form of p is then asserted into the database. This new form has an extra argument added to the beginning of the head and to each subgoal in the body of a clause. For instance, in the case of p above, the new clause asserted would be

```
p(TheoryIn) :- guard(TheoryIn,[0 | _],[]).
```

guard does the real work in meta. It makes sure that the clause can be used in TheoryIn, the theory descriptor handed by demo to the call of p. The second argument of guard describes the theories in which p is known by giving the initial sequence of all theories that can access p. For each theory that cannot access p, the third argument of guard contains the initial sequence of all theories where the clause is no longer valid. This third argument will be a list of lists. A clause with aguments and subgoals, such as

```
p(A) :- g(A,B), r(B)
```

would be asserted into theory [0,1] as

```
p(TheoryIn,A) :- guard(TheoryIn,[0,1 | _],[]), g(TheoryIn,A,B),r(TheoryIn,B).
```

The theory behind guard is simple. If a sequence of addTo's is done, the theory descriptor coming out of the last addTo is a list of that theory. For example,

```
:- addTo([],p,T1),addTo(T1,q,T2),addTo(T2,r,T3)
```

would instantiate T1 to [0], T2 to [0,1], and T3 to [0,1,2]. p is known in all of these theories, and the second argument of it's guard clause is [0 | _], which would unify with any of the above theory descriptors (T1,T2,T3). [0 | _] is said to be an *initial sequence* of these theory descriptors. So, any theory descriptor starting with a 0 knows about p. The predicate q's initial sequence is [0,1 | _]. This is not an initial sequence of [0], so q cannot be found in the first theory, but will be found in theories [0,1] and [0,1,2].

dropFrom works by adding to the third argument of a clause. To continue the above example, if a dropFrom([0],p,T4) is executed, T4 will be instantiated to [0,3], and the clause in the database will be changed to

```
p(TheoryIn) :- guard(TheoryIn,[0 | _],[[0,1,2 | _]]).
```

So, this third argument contains a list of initial sequences of theory descriptors with roots where p was defined (theory [0] in this case), where p no longer exists. If p was put into meta from another addTo, this addTo would create a different clause for p of the same form as above. The guard clause, if the first two arguments unify, will check TheoryIn against the theory descriptors in the list in the third argument position, and fail guard if any are an initial sequence of TheoryIn.

guard also handles the union of theories. If the user executes

```
demo(T1+T2+...+Tn,Goals),
```

the entire group of

```
T1+T2+....+Tn
```

is passed through to the Goals in the first argument. If guard notices a union, it will try and find the clause in the first theory in the list. If it isn't there, the next theory will be tried. This checking is put into guard to avoid excess work. If a late subgoal in the list of Goals fails to be found in T1, the interpretor must not fail all the previous subgoals just because this subgoal can't find anything in T1. So, this subgoal will look in T2 next.

demo(TheoryID,Goals) will call Goals with a particular TheoryID. The list of goals is rewritten to pass in TheoryID as the first argument. If TheoryID is a variable, demo will unify TheoryID to a theory descriptor where the goal is true. addTo and dropFrom are implemented so that they can backtrack. If a cut is executed, when backtracking is done, the clauses will be left in the database. This is not a problem, however, since the guard for these clauses will never fire again.

There are two extra builtins which have been added. One is called

baseTheory(FileName),

which is like consult. The file FileName should contain lines like

```
    theory(TheoryName).
    theory for TheoryName.....
    ......
    endTheory.

    theory(NewTheoryName).
    etc.
```

Once a theory is described, clauses can also be added by

    TheoryName :: Clause.

To use clauses in a theory entered using baseTheory, the predicate

theory(TheoryName,TheoryID)

is used in conjunction with demo. For instance,

    ......, theory(phideaux,T1), demo(T1,Goal), ...

would pick up the theory ID bound to the name phideaux and use it in demo. These theory names cannot themselves be used by the core meta predicates. An example of a file for baseTheory follows.

```
------------

theory(a).
a(a).
a(b).
a(c).
a(d).
endTheory.

theory(b).
b(a).
b(b).
b(c).
b(d).
endTheory.

b :: b(e).
a :: a(e) :- a(b).

-------------------
```

6-B.4.5 Modification of the Abstract Prolog Machine

The fundamental work during this year was an extensive study of methods of modifying and extending the Abstract Prolog Machine underlying the basic Prolog compiler system to cater to the needs of the extended metaProlog compiler system. This involved the following difficult issues, among others:

a) Explicit, efficient methods for representation of the theories (i.e., program databases).

b) Methods for representing the historical course of the growth of new theories from old by the addition and subtraction of axioms (necessary for providing backtracking, which is in turn necessary for a semantically sound implementation).

c) Methods of moving program clause code out of the static code area and instead, storing it on the dynamic global heap of the Abstract Prolog Machine (this being necessary for the representation of theories). Of particular concern is the basic issue which arose during the earlier study of interpreter-based approaches to metaProlog, namely the problem of being able to sometimes view clause code as program code directing machine execution, and sometimes being able to view it as structured term objects which can be unified with other Prolog terms. In particular, our goal is to eventually develop methods of representing partially specified or partially constructed formulas and theories which are refined and further instantiated in the course of deductive computation.

d) Methods of implementing "black box" objects (really a kind of abstract data type) which have clear external "Prolog" structure from the point of view of the Abstract Prolog Machine, but which may "hide" their true structure behind various C routines (or assembler, etc.). These will eventually be used for such problems as representing parts of the internal state of the Abstract Prolog Machine as Prolog-type objects, or for representing a the tuples of a relational database table as a set of facts for some predicate, together with maintaining the necessary associated external buffers, etc.

e) Methods for modifying the Abstract Prolog Machine "trail" to support more generalized backtracking schemes. (The ordinary Prolog "trail" simply resets variables bound by unification to "undefined". The generalized trail must not only support resetting variables to previous defined values, but must also support the invocation of internal black box [abstract data type] routines; e.g., the flushing of buffers associated with database access objects, etc. This methodology is called "event trailing".)

The work conducted during the design phases of the year yielded a number of designs and techniques for coping with the concerns a)-e). The work of this phase was to make an integrated set of design choices and rebuild the basic Prolog compiler system to implement those designs. The basic decisions, which interlock in a number of ways, were as follows:

i) The direct physical representation of individual clause code is unchanged, but the clause code is no longer allocated in a separate code area, but is allocated on the dynamic heap. Because of the (eventual) needs of garbage collection (not yet implemented), code blocks now must be delineated by headers and footers, thus supplying the garbage collector (really a sliding compactor) with appropriate information about the block.

ii) The problem of the dual views of compiled code has only partially been dealt with in the present implementation, though we have design principles which should solve the entire problem For the moment we are not providing the ability to manipulate compiled clauses by direct unification. (However, they can be indirectly manipulated via the efficient decompilation technique which underlies the implementation of the *dropFrom* and *axiomOf* built-in predicates.) The problem would be less difficult in an interpreter-based approach, since the differences between the internal representation of clauses as formulas and the standard representations of terms is not terribly great. But in the compiler-based approach, the differences are great. However, the fast decompilation techniques developed for the basic Prolog compiler provide a basic method of unifying compiled formulas with term-type representations of the same formulas. The use of the decompilation techniques in the Prolog compiler has a slightly specialized focus, and so some work must be done to extend them to the fully general setting. These techniques do not immediately solve the remaining problem, namely that we must be able to treat two compiled formulas as terms and unify them. We believe that a somewhat more radical use the ideas involved in the fast decompilation techniques will solve the problem, but some investigation remains. [In principle, the problem is solvable: Simply decompile the two formulas, and then unify the decompiled results. However, this appears to be too inefficient. Also, the presence of uninstantiated interpreter variables in compiled formula structure, as opposed to user (quantified) variables, may cause some complications which must be dealt with.]

iii) Theories are represented by collections of indexing instructions controlling access to individual clause code. The indexing instructions are a subset of those used in standard Prolog compilers. We simply make different use of them in this setting.

iv) We chose to implement two different technical approaches to the problem of maintaining the historical relationships of construction (i.e., new theories or databases from old), thus in fact yielding two slightly different metaProlog compilers. One is based on hash tables, the other on a high-level indexing instruction approach. Fundamentally, they provide the same facilities, but we are interested in exploring possible performance differences.

v) We developed low-cost methods of distinguishing trail entries, between standard entries ("reset to undef") and event entries ("reset to a previous value", or "run some strange attached procedure when asked to reset"). Thus the system provides completely general event trailing.

vi) We have partially implemented "black box" abstract data types, but not in the full generality which we seek. In particular, work remains to integrate them fully with the system unifier.

We now have the two alternative prototype metaProlog compilers running under Berkeley UNIX 4.2 on a DEC VAX 780. They both perform quite efficiently: When running the standard Prolog naive reverse benchmark as standard Prolog code once the context (theory) is fixed, they perform at about 9K LIPS. Thus the overhead due to the metaProlog facilities is at most 10%.

6-B.5 DBMS Interfaces

It is common that many envisaged and intelligent systems will need to work with large knowledge bases. In some such cases, the set of non-trivial rules will be quite large, and technology to efficiently manage very large rule sets must be developed. However, in many more cases, the size of the rule set will be moderate, but nonetheless, the set of facts (improper rules with no conditions or bodies) will be extremely large. In some applications, these sets of facts will be built especially for the intelligent application and will be utilized only by that application. In this case, the facts can be stored and managed by whatever method is most suitable and efficient for the rule processor (in our case, a logic-programming processor). But in many cases, the set of facts exists in large traditional DBMS-managed databases which have been built up prior to construction of the intelligent system. Moreover, the organization owning the databases already has many applications running over those databases. If the database is static, it may be possible to produce a new version crafted especially for the use of the intelligent system. However, in very many instances, the database is dynamic and it is important for the intelligent system to be working with current data. In such a case, the only possible solution is for the intelligent system to talk directly to the DBMS-managed database.

In order to attack this latter problem, and to begin the exploration of coupling logic-based systems to very large sets of facts, we initiated a study directed at the construction of direct interfaces between our basic Prolog compiler and traditional relational DBMS-managed databases. Since we had the source code for the Berkeley version of INGRES available on our VAX 780, we chose this as our target.

Like most DBMSs, the foreign-code interface to INGRES is much less than would be desired for this application. In essence, the interface requires the foreign program to construct a string representing the query, the DBMS then parses the string, processes the resulting query, and returns a file containing all answers to the query. The need to construct the query as a string which is then parsed by the DBMS is a clear efficiency bottleneck. However, it is clear that in cases where greater efficiency is strongly demanded, it should be possible to arrange with the implementers of the DBMS to provide a more sophisticated interface which would perform much more efficiently. All that is needed is for the foreign program (e.g., a logic-programming system) to be able to construct and pass to the DBMS an "internal" form of the query which is identical to what the DBMS produces as a result of its parsing. Since logic systems are symbol manipulators par excellance, it is easy for them to produce such internal forms. (Additional built-in primitives will probably have to added to allow the logic system to create the precise data structures utilized by the DBMS for its internal forms.) Of course, it will be the responsibility of the program on the logic side of this communication pipe to see to it that a correct internal DBMS query is produced.

At first glance, it would appear that there is a serious mismatch between a Prolog-based system's tuple-at-a-time approach to computation and the DBMS's set-oriented production of all solutions to queries. However, we turned this apparent mismatch to our advantage. Since we were running under UNIX, we created a third "buffer" process to manage the communication between our Prolog and INGRES. This buffer process passes the query from Prolog to INGRES, and accepts back from INGRES the file of solutions. The buffer process then passes the tuples from the answer set to Prolog one at a time as they are needed, maintaining backtracking pointers,etc. This also overcomes another deficiency of INGRES, which will only interact with one foreign process at a time. The buffer process allows a single or multiple Prolog processes to have several queries against the database active at the same time. Also, we constructed the interface in such a way that a single Prolog process can be "talking to" more than one database at the same time (even managed by different DBMSs) via multiple buffer processes. Thus, an intelligent system can access data from multiple databases, perform joins across distinct DBMS databases, etc.

6-B.6 Future Plans

In the near term, we will continue polishing the two metaProlog compiler systems, adding some simple missing facilities and of course searching for bugs as we exercise them on moderately simple problems. Then we will embark on what we regard as Phase III of the metaProlog project, which will involve the following general tasks and concerns. [Undoubtedly, the approach will be somewhat iterative and cyclic in character: Code up the experiments, modify or extend the system according to results, recode or retest old experiments, code new experiments, etc.]

1) Port the system to the SUN 3 workstations which have been purchased for the laboratory under a DOD URIP grant.

2) Explore the use of the system (or its further extensions and refinements) for some of the more sophisticated programs and knowledge representations developed during earlier project stages (and subsequently, up to the present). Serious attempts will be made to develop programs of substantial scale and capabilities, possibly rebuilding existing systems (such as KNOBS/KRS, etc.) utilizing the new techniques. This latter approach has certain obvious advantages in not having to develop new applications knowledge, and in allowing reasonable comparisons to the old system(s).

3) Develop the use of the abstract data type (ADT) mechanism to provide virtual interfaces to large DBMS systems (cf. Section 6-B.5 below), both of existing types and of new types (utilizing both software and hardware support) to be developed (e.g., by Prof. P.B.Berra's sister NAIC project). The simplest use is to simply provide virtual representation of DBMS relational tables as sets of ground facts through such ADTs. A more sophisticated approach is to allow the Prolog compiler to perform various optimizations which might involve replacing compound subformulas by a single virtual formula which is determined by an ADT. The ADT is defined by passing the compound subformula to the DBMS as a query to be processed. (This might be passed in an optimized form below the normal program interface level.) This would allow the DBMS to apply its optimization algorithms, perform joins, projections, etc.

4) Extend the system to support partially instantiated formulas and theories, and explore methods of providing alternative control facilities.

Bibliography

Bowen, K.

[1985] *Meta-level programming and knowledge representation,* **New Generation Computing**, v.3 (1985), pp.359-383.

[1986a] *Meta-Level Techniques in Logic Programming,* **Proc. Artificial Intelligence '86 Conf.**, Singapore, March, 1986.

[1986b] *New Directions in Logic Programming,* **Proc. 1986 ACM Annual Computer Science Conference**, Cincinnati, 1986.

Bowen,K., Buettner, K., Cicekli, I., and Turk, A.

[1986] *The design and implementation of a high-speed incremental portable Prolog compiler*, **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986, pp. 650-656.

Bowen, K. and Kowalski, R.

[1982] *Amalgamating language and metalanguage in logic programming,* in **Logic Programming**, eds. K.L. Clark and S.-A. Tärnlund, Academic Press, London and New York, 1982, pp. 153-172.

Bowen, K. and Weinberg, T.,

[1985] *metaProlog: A metalevel extension of Prolog,* **Proc. 1985 Symp. on Logic Programming,** Boston, 1985, pp. 48-53.

Buettner, K.

[1986] *Fast decompilation of compiled Prolog clauses*, **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986, pp. 662-668.

Fain, J., Gorlin, D., Hayes-Roth, F., Rosenschein, S.J., Sowizral, H., and Waterman, D.

[1982] **Programming in ROSIE: An Introduction by Means of Examples**, Tech.

Report N-1646-ARPA, Rand Corp., Santa Monica, CA, 1982.

Hayes, P.J.

[1977] *In defense of logic,* IJCAI 5, 1977, pp. 559-565.

Hayes-Roth, F., Waterman, D., and Lenat, D.

[1983] **Building Expert Systems**, Addison-Wesley, Reading, MA, 1983.

Kuipers, B.J.

[1975] *A frame for frames: representing knowledge for recognition,* in **Representation and Understanding**, eds. D.G. Bobrow and A. Collins, Academic Press, New York, 1975, pp. 151-184.

Mellish, C.S.

[1982] *An alternative to structure sharing in the implementation of a Prolog interpreter,* in **Logic Programming**, ed. K.L.Clark and S.-A. Tärnlund, Academic Press, London and New York, 1982, pp. 99-106.

Minsky, M.

[1975] *A framework for representing knowledge*, in **The Psychology of Computer Vision**, ed. P.H. Winston, McGraw-Hill, New York, 1975, pp. 211-277.

Pereira, L.M., Pereira, F.C., and Warren, D.H.D.

[1978] **User's Guide to DECsystem-10 PROLOG**, Dept. of Artificial Intelligence, University of Edinburgh, 1978.

Turk, A.

[1986] *Compiler optimizations for the WAM,* **Proc. 3rd Int'l Logic Programming Conf.**, London, 1986, pp. 656-662.

Warren, D.H.D.

[1980] *An improved PROLOG implementation which optimises tail recursion,* in **Logic Programming Workshop** Participants' Proceedings, 1980, pp. 1-11 (revised version appeared as *Optimizing tail recursion in Prolog,* in **Logic Programming and Its Applications**, ed. M. van Caneghem and D.H.D. Warren, Ablex Pub. Co., Norwood, NJ, 1986, pp. 77-90.)

[1977] **Implementing Prolog — Compiling Predicate Logic Programs**, Dept. of Artificial Intelligence, University of Edinburgh, Research Reports 39 & 40.

[1983] **An Abstract Prolog Instruction Set**, SRI Technical Report, 1983.

Warren, D.H.D. and Pereira, L.M.

[1977] *PROLOG: the language and its implementation compared with LISP,* in **Proceedings of the Symposium on Artificial Intelligence and Programming Languages**, SIGPLAN Notices, v.12, No. 8 (August) 1977, and SIGART Newsletter No. 64 (August) 1977, Association for Computing Machinery, New York, 1977, pp.109-115.

Appendix 6-B-A

The Design and Implementation of a High-Speed Incremental Portable Prolog Compiler

Kenneth A. Bowen, Kevin A. Buettner, Ilyas Cicekli, Andrew K. Turk

Logic Programming Research Group
School of Computer & Information Science
Syracuse University
Syracuse, NY, 13210 USA

Abstract

The design and implementation of a relatively portable Prolog compiler achieving 12K LIPS on the standard benchmark is described. The compiler is incremental and uses decompilation to implement retract, clause, and listing, as well as support the needs of its four-port debugger. The system supports modules, garbage collection, database pointers, and a full range of built-ins.

1. Introduction

In the course of exploring implementation techniques for metalevel extensions of Prolog (cf. Bowen and Kowalski [1982], Bowen and Weinberg [1985], Bowen [1985]), it became apparent that a fast flexible Prolog system would be a useful tool to serve as a starting point for developing experimental implementations of the extended Prolog systems. In late 1984, we planned to base the system on the designs of Warren [1983], implementing a byte-code interpreter for the abstract machine in C, while implementing the compiler itself in Prolog.

We were fortunate to join forces with the group working at Argonne National Laboratory (Tim Lindholm, Rusty Lusk, and Ross Overbeek) which was interested in the implementation of Prolog on *multiprocessor machines*. They had already written a portable byte-code interpreter in C on a VAX 780. Unfortunately, the performance of the Argonne WAM was disappointing. The naive reverse benchmark (nrev) ran at only about 4K LIPS. We concluded that the relatively slow speed was due to a combination of the portability requirements and the data structures necessary for multi-processor implementation.

The disappointing performance of the Argonne WAM, coupled with an interest in implementing a Prolog system on 68000-based machines, led Turk to begin exploring a new implementation of a byte-code interpreter written in C, while as a group we continued work on the compiler.

In late February of 1985 we had a first version of the compiler itself operational in C-Prolog. Unfortunately, we found ourselves hampered by C-Prolog's restricted memory size and apparent lack of significant tail recursion optimization and garbage collection. The compiler itself had grown fairly large, reflecting our explorations of various optimization techniques. When we began to boot the compiler on itself, we were frustrated to discover that our compiler immediately overran the maximum allowable local and global stack spaces.

This work supported in part by US Air Force grant AFOSR-82-0292 and by US Air Force contract F30602-81-C-0169. The authors are very grateful to the following people for numerous valuable conversations on the topics of this paper: Hamid Bacha, Aida Batarekh, Keith Hughes, Jim Kajiya, Kevin Larue, Jacob Levy, Tim Lindholm, Rusty Lusk, Joe Mills, Hidey Nakashima, Ross Overbeek, Karl Puder, and Toby Weinberg.

At this time, Buettner, in a one month burst of enthusiasm, roughed out a new byte-code interpreter for the abstract machine coupled with an implementation of a moderately sophisticated compiler, all written in C. We now found ourselves in the (perhaps enviable) position of possessing three distinct implementations of the abstract machine (all written in C) and two compilers, one written in C and the other in Prolog.

While there were some differences in structure between the compilers, they both operated on basically the same principles. On the other hand, our two home-grown implementations of the abstract machine appeared to use significantly different techniques, and of course differed markedly from the Argonne implementation. Since both of our local WAMs executed nrev at better than 6K LIPS and both authors asserted that not all opportunities for optimization had been exploited, we decided to pursue development of both machines and compilers in parallel.

In the course of the summer of 1985, both machines evolved towards a common structure, and began achieving speeds in nearing 10K LIPS on nrev. We also had the interesting experience of booting the Prolog-based version of the compiler using the C-based Prolog compiler. Both of the Syracuse abstract machines were evolving towards a common structure, so we chose Buettner's system as the basis for the rest of our work.

We completed the Prolog-based version of the compiler and delivered a copy to the Argonne group in late August '85. It is expected that this will be made publicly available along with the Argonne WAM sometime in the near future. The rest of this paper will describe the design, structure, and facilities of the C-based system. We will assume familiarity with Byrd, Pereira and Warren [1980], Pereira, Pereira, and Warren [1978], and Warren [1983].

2. Organisation of the System

To the user, our system has the appearance of a standard Edinburgh-style interactive interpreter. However, it is really an incremental compiler. Thus we have no need to support a separate interpreter with all the difficulties of consistency between a compiler and an interpreter. Briefly, the major services provided by the system are as follows:

• The compiler is resident in the system, incrementally compiling original and added program clauses (including those added by assert) as well as goals.

• Programs may be organized into modules which are independent of file structure in that multiple modules may be included in a single file (a single module can also be spread over several files). Visibility of procedures is controlled by use of import/export declarations. Clauses not appearing within a module declaration are stored in a default global module. Constants and functors are globally visible.

• Garbage compaction of the global stack (heap) and trail is provided using a pointer-reversal algorithm of Morris[1978]. No garbage collection is provided for the code space.

• Run-time use of retract, clause, and listing is accomplished via a general decompilation technology (described in detail in Buettner [1985]). This technology is also used to support the debugging subsystem.

• A full four-port debugging model (cf. Byrd[1980]) is provided. It relies on the decompilation technology mentioned above and accomplishes its task by constructing linked lists representing the local stack frame entry and exit on the global stack (heap). It is largely complete, though some standard commands remain to be implemented.

• Database pointers are supported. These exist as Prolog terms which can occur in other terms and predicates.

The system supports the full range of built-ins standard in Edinburgh-style Prolog systems. Some are implemented in C, with the rest being written in Prolog and compiled by the system.

The code for the system occupies approximately 135K bytes of virtual memory (76K bytes of physical memory) when loaded. The machine will run nrev at 12K LIPS on a list of length 100. Nrev reverses a 1000 element list at 10.5K LIPS. The lower speed reflects the time spent in garbage collection.

3. Compilation

The Prolog-based compiler includes a lexical analyzer and DCG parser to read in Prolog source code. This is necessary because systems like the Argonne WAM do not have a built-in reader. The parser performs its variable analysis on the fly, and produces a term which represents a clause. Another pass translates the clause representation into a sequence of abstract machine instructions. Uninstantiated logical variables are used to represent code addresses inside the compiler.

During the second pass, the intermediate code for the individual clauses constituting a procedure is connected by indexing instructions. Our method of indexing, which differs from Warren [1983], will be described later. The output of the second pass is a complex Prolog term representing the procedure. Assembly amounts to a traversal of this term, calculating symbolic addresses as necessary, and linearizing the entire structure. Loading is then straight-forward.

The C-based version of the compiler utilizes a standard Prolog reader to read the clauses as terms. It makes one pass through the term, performing its variable analysis and building appropriate tables. On a second pass through the term, it generates and loads the instructions for the clause, linking them into the naive try-me-else indexing chain for the procedure. Full indexing for the procedure is generated when the module containing the procedure is sealed.

Both compilers optimize argument register usage and can re-order certain operations to streamline the resulting code. In addition, a small amount of "clause lookahead" is used so that variables will end up in the correct registers for the first subgoal. Argument registers are reused whenever possible.

4. Indexing

The indexing code for a procedure has two tasks to accomplish. When the indexing argument (normally A0) references an unbound variable, the machine must attempt each clause of the procedure in the correct order. If the indexing argument dereferences to a non-variable, the indexing code should select *only* those clauses which could possibly match, while excluding the majority of the useless clauses which would fail.

We have not modified the indexing instructions of Warren [1983], but we do employ them in a different manner. Focusing on the first argument of a procedure, a *block* of clauses is a maximal subset of the clauses for the procedure, contiguous in the given clause ordering, all of whose first head arguments are of the same type, either constant, compound term (other than list), variable, or list. The compiler uses indexing instructions at the lowest level to control access to each block, with second-level indexing instructions to control transfers between blocks.

Try chains and *try_me chains* define linked lists of code patches that the machine will traverse at runtime. These code patches can either be individual clauses, blocks of clauses governed by switch instructions, or nested *try chains*. A *try chain* consisting of try, retry and trust instructions is used to link clauses together when the indexing code itself must be physically seperate from the clause code. A *try_me chain* interleaves the clause code and the indexing code by using the try_me_else, retry_me_else and trust_me instructions.

Our indexing method creates one *try_me chain* that links all the clauses of a procedure together in the right order. This chain is used when the indexing argument references an unbound variable. *Try_me chains* are simple to compile and easy to change after an assert or retract.

When the indexing argument contains a non-variable, the machine will use a *try chain* to access clauses. Access to clauses in this chain will be controlled by switch instructions to eliminate useless clauses. Since all *try chains* are physically seperate from the clauses they govern, the system can rewrite the non-variable indexing for a procedure without recompiling the clauses in the procedure.

While using two seperate indexing chains uses more instructions than the method presented in Warren [1983], the same number of choicepoints are created at runtime. The extra flexibility afforded by seperate indexing chains is worth a few more instructions in the code space. Figures 4.1 and 4.2 schematically indicate the structure of our scheme.

5. Cut

For the most part, we have implemented the instruction set of Warren [1983] with only minor modifications. The most significant extension to date is the addition of a new machine register, called *cutpt*, and new instructions to manipulate it. These instructions and their effects are listed below. The last instruction is only necessary in the compilation of soft cuts.

Instruction	Action
set_B_from_cutpt	B := cutpt
set_B_from Yn	B := Yn
save_cutpt_in Yn	Yn := cutpt
save_B_in Yn	Yn := B

Figure 5.1. Instructions Necessary for Cut.

The problem with cut is that the compiler cannot always know how many choice points will be created for a procedure containing a cut. Consider the following trivial program, whose source consists of the two facts f(a) and f(b):

```
f/1:    switch_on_term      C1a,L1,fail,fail
L1:     switch_on_constant  2,[a:C1, b:C2]
C1a:    try_me_else         C2a              % f(
C1:     get_constant        a,A0             %   a)
        proceed                              %   .
C2a:    trust_me_else       fail             % f(
C2:     get_constant        b,A0             %   b)
```

```
        proceed                              %
```

When the first clause C1 is executed, there can be either zero or one choice points for the procedure f/1. The compiler cannot detect which is the case because it depends on the incoming value in the first argument register A0. If the incoming value in A0 is the constant *a*, there will be no choice point created for the procedure f/1, but if the incoming value in A0 is an unbound variable, there will be one choice point created for the procedure f/1. Therefore, cutting a procedure is not as simple as removing the topmost choicepoint.

The new register *cutpt* is treated in the abstract machine as follows. The value of the B register is automatically stored in the cutpt register by a *call* or an *execute* instruction. This records the address of the last choice point before the next procedure is invoked. The current value of the cutpt register is saved in each choice point along with the other machine registers.* The cutpt register is reset from the value stored in the last choice point when backtracking occurs.

The cutpt register contains the address of the last choicepoint created before entering a procedure. By simply moving this value into the B register, the machine can remove all the choicepoints created by the current procedure, no matter how many there were. Since cutpt is modified by every subgoal, the compiler must store the cutpt in a permanent variable to preserve it across a call.

The following examples illustrate how the compiler uses these instructions to compile cuts.

Example 1: p :- q1, !, q2.

```
    allocate            1
    save_cutpt_in       Y0
    call                q1/0,1
    set_B_from          Y0
    deallocate
    execute             q2/0
```

Example 2: p :- !.

```
    set_B_from_cutpt
    proceed
```

Example 3: p :- q1, q2, !.

```
    allocate            1
    save_cutpt_in       Y0
    call                q1/0,1
    call                q2/0,1
    set_B_from          Y0
    deallocate
    proceed
```

This approach can be optimized.

6. Conclusions

The abstract machine design of Warren [1983] together with the compilation techniques suggested by his examples are a sound piece of software engineering. We have filled in some gaps such as the implementation of cut which were omitted in his discussion, and have introduced modifications in the pursuit of refining and optimizing performance. The present system provides an excellent basis for our primary goal, the pursuit of implementations of meta-level Prolog systems. Our approach will be to introduce modifications to the abstract machine providing the required functionality, the primary one being a change in the treatment of the code space. This will be coupled with appropriate changes in the compilers. We expect this to lead to efficient implementations of the experimental systems.

7. References

Bowen, K.A., and Kowalski, R.A., Amalgamating language and metalanguage in logic programming, in *Logic Programming*, ed. K. Clark and S.-A. Tarnlund, 1982, pp 153-172.

Bowen, K.A., and Weinberg, T., A meta-level extension of Prolog, *1985 Symposium on Logic Programming*, Boston, IEEE, 1985, pp. 48-53.

Bowen, K.A., Meta-Level programming and knowledge representation, *New Generation Computing*, 3, 1985, pp. 359-383.

Buettner, K.A., Decompilation of compiler Prolog clauses, submitted.

Byrd, L., Prolog debugging facilities, in Byrd, Pereira, and Warren, 1980.

Byrd, L., Pereira, F., and Warren, D., *A Guide to Version 3 of DEC-10 PROLOG*, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1980.

Morris, F.L., A time- and space-efficient garbage collection algorithm, *Communications of the ACM*, 21, (1978), pp. 662-665.

Pereira, L.M., Pereira, F.C., and Warren, D.H.D., *User's Guide to DECsystem-10 PROLOG*, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1978.

Warren, D.H.D., An abstract Prolog instruction set, SRI Technical Report, 1983.

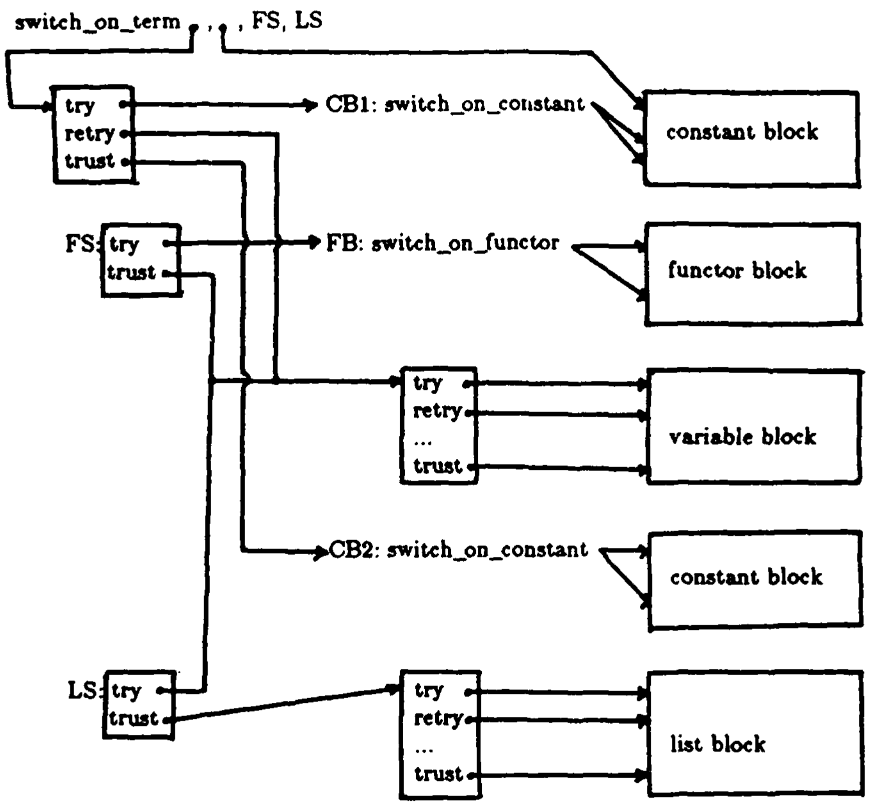

Figure 4.1. Overall indexing structure.

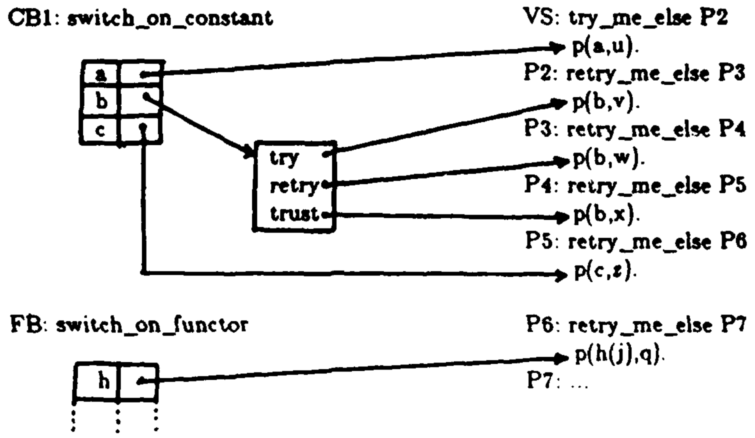

Figure 4.2. Detail of indexing structure.

Appendix 6-B-B

Fast Decompilation of Compiled Prolog Clauses

Kevin A. Buettner

Logic Programming Research Group
School of Computer & Information Science
Syracuse University

ABSTRACT

Serious Prolog implementations in recent years have been primarily compiler-based, nearly all of which are founded on the abstract instruction set of Warren [1983]. The performance achieved by such implementations greatly outstrips that attainable in interpreter-based systems. Unfortunately, the sophistication of these compiler-based environments is often inferior to environments of interpreter-based systems to the extent that a "compatible" interpreter is often required for serious software development. Among the deficiencies of these environments, database operations such as assert, retract, and clause seem to be particularly afflicted. In addition, the ability to debug compiled code has been either non-existent or at best, very constrained.

Unlike compiler technology of many traditional languages, there is little reason for this to be the case in Prolog. An efficient implementation of retract, listing, and clause by decompilation of compiled clause code is the subject of this paper. Techniques used in the implementation of the decompilation process have also proven useful in the implementation of the standard four port debugger found in many Prolog systems.

Introduction

The programming language *Prolog* has a number of builtin predicates for operating on the global database. Among these operations, the predicates *assert*, *retract*, and *clause* are found in most Prolog systems. Due to their non-logical nature, these predicates have gained a certain notoriety in the logic programming community. In spite of the bad press that these operations have gotten, many practical and useful programs are written which need and require these operations although in theory it is possible to dispense with them.† The view of the author is that these operations are important and should be provided in modern Prolog systems. Even languages which claim a sounder logical foundation with respect to the database operations‡ have implementation problems similar to those found in ordinary Prolog.

The first Prolog environments were interpreter based. The database operations are relatively easy to implement in these environments due to the similarity between the run-time data structures and the code structures. At the present, the trend seems to be towards iler-based Prolog environments. Most cur.ent compiler-based implementations have

his work supported in part by U.S. Air Force contract number F30602-81-C-0169.

† It is often possible to reorganise programs so that assert and retract are not used, but at the expense of efficiency and quite often clarity of expression.

‡ e.g, metaProlog (cf. Bowen and Weinberg [1985]) provides the operations addTo and dropFrom which are used to create new theories from old ones.

their roots in the abstract machine of David Warren [1983]. With the advent of this compiler-based technology, implementation of the database operations has become more challenging due to the fact that the run-time data structures are quite a bit different from the code structures. The run-time data structures are much the same as they were under interpreter-based implementations, but the code structures are different; they are now sequences of instructions to execute. Implementation of the *assert* operation with this new technology isn't too difficult. All one needs to do is invoke the compiler on the clause to assert, obtain the sequence of instructions and link these into the indexing scheme.

It is the implementation of either retract or clause that is more interesting. These operations are similar in that we must be able to take fairly arbitrary patterns and find a clause in the database which matches these patterns. Again for interpreter-based systems, this isn't usually very hard due to the similarity between the run-time structures and the code structures. But matching clauses in a compiler-based system is more difficult. As observed by Clocksin [1985] in his discussion of the implementation of Prolog-X, there are basically three ways to organize the database so that this matching may be performed:

1) Save a copy of the source clause in addtion to the compiled clause. Clocksin points out that many compiler-based LISP systems use this technique. But for a number of reasons, this technique is unsatisfactory for Prolog.
2) Keep only the compiled clause code in the database and decompile it when matching needs to be performed. Performing the decompilation efficiently is the subject of this paper.
3) Compile and insert into the database a unit clause which relates the source structure with the database reference of the real clause code. This is the method that Prolog-X uses. Clocksin notes that it enjoys the advantage that clause searching and matching are done by the usual mechanisms of the abstract machine. It has the disadvantage that compilation takes approximately twice as long since each clause is compiled twice and approximately twice as much space is required in the database.

This paper describes a way of running certain of the abstract machine instructions in a different "mode" causing a clause to decompile itself. Very few limitations exist on the types of compiler optimizations that may be performed. Even for extremely optimized code, however, there are ways to augment the output of the compiler so that the methods described will still work. In the ensuing discussion of the decompilation technique, familiarity with the abstract machine described by Warren [1983] is assumed.

Decompilation

Suppose we have a source clause of the following form:

$$H \text{ :- } G_1,\ldots,G_m .$$

This translates (schematically) to:

```
match args of H
set up args of $G_1$
call $g_1/n_1,ES_1$
set up args of $G_2$
call $g_2/n_1,ES_2$
.
.
.
set up args of $G_m$
execute $g_m$
```

In the above, call and execute are actual instructions. *match* represents the sequence of get and unify instructions that perform the head matching. *set up refers* to a sequence of put and unify instructions that install arguments for the call or execute instructions. In practice, the initial *match* and *set up* instructions are often merged as an optimization. This assumption will not affect the process which we are about to describe.

Before discussing the nitty gritty details, it should be noted that a fair amount of structure is created by the object code in normal execution. For example, if the clause is run with all uninstantiated arguments, by the time of the first call the original variables will be instantiated to structure representing the arguments in the head of the clause. Similarly, just before a call or execute, the A registers contain the arguments of the goal about to be called. To get back the structure of the entire clause entails taking these pieces and wrapping them up with the appropriate functors that represent the head, goals, commas between the goals, and the ":-" between the head and the goals. The decompilation technique will be discussed in two stages. The first stage is to look at an object code transformation that when run by the underlying Prolog engine will give back the original clause structure. The second stage is rather easy; ways to avoid *doing* the actual transformation are considered.

A transformation that will take object code representing a clause (in the above form) and produce object code which can be run with a single argument (which may have varying levels of instantiation) and return the structure (or pieces of the structure as the level of instantiation requires) will now be described. The idea is to tack on a prologue to the beginning of the code which will create the uninstantiated variables for the head and set up the clause predicate name. The prologue will create something of the form

$$h(Arg_1,Arg_2,\ldots,Arg_{n_0}) \text{ :- } G$$

The A registers will have pointers to the variables $Arg_1 \ldots Arg_{n_0}$ respectively. Because the A registers contain pointers to variables, the get instructions in the *match* section of the clause code will be forced into write mode and will create structure. We don't want the call and execute instructions to run, so we replace them with code which will unload the argument registers and produce the appropriate goal structures.

Schematically, the transformation looks like this:

Original Code	Transformed Code	Comments
	get_structure :-/2, A1	Prologue
	unify_variable As	
	unify_variable At	
	get_structure h/n_0,As	
	unify_variable A1	
	.	
	.	
	.	
	unify_variable An_0	
match args of H	*match* args of H	
set up args of G_1	*set up* args of G_1	
call $g_1/n_1,ES_1$	get_structure ','/2, At	Call
	unify_variable As	Transformation
	unify_variable At	
	get_structure g_1/n_1,As	
	unify_local_value A1	
	.	
	.	
	.	
	unify_local_value An_1	

.		
.		
.		
set up args of G_m	*set up* args of G_m	
execute g_m	get_structure g_m /n_m ,At	Execute
	unify_local_value A1	Transformation
	.	
	.	
	unify_local_value An_m	
	proceed	

Notes:

- The As and At in the above transformation refer to unused temporary (argument) registers.
- unify_local_value instructions are used to insure that pointers in structure built on the heap also point to objects on the heap. This is necessary due to the fact that an argument register may have a pointer to a variable on the local stack. If a unify_value instruction is used in this situation, a pointer from the heap to the local stack is installed causing a dangling reference when the environment is deallocated.

The above transformation scheme will work for clauses with at least one goal where both the head and each of the goals has at least one argument. As it stands, it will not work for unit clauses or for clauses which have nullary goals. These cases, however, are not difficult to handle. For unit clauses, the prologue changes slightly. When it is necessary to work with a nullary goal, a get_constant instruction is used instead of a get_structure instruction. For example, the prologue for a unit clause whose head has no arguments is:

```
get_constant  h,A1
```

The prologue for a unit clause with one or more arguments is

```
get_structure  h/n0,A1
unify_variable  A1
 .
 .
 .
unify_variable  An0
```

An execute instruction with no arguments translates to

```
get_constant   gm /nm ,At
proceed
```

It is left as an exercise for the reader to fill in the other variations. It is possible to define the translations so that there are no variations between unit clauses and rules, but in practice these variations cause little difficulty. The translation procedure given here has the added advantage that in attempting to match (unmatchable) partially instantiated clauses, failure can occur quite early. To make the translation procedure more uniform, it would be necessary to delay building the structure for the :- until the execute instruction. It would also be necessary to translate proceed instructions, which are currently untouched.

As an example, consider the clauses that make up part of a popular benchmark:

```
nrev([],[]).
nrev([H|T],L) :- nrev(T,RT), conc(RT,[H],L).
```

This compiles to the following object code:

```
nrev/2:
            switch_on_term        L434,
                                  L440,
                                  L464,
                                  fail
L434:       try_me_else           L458,2 %
L440:       get_nil               A1                 % nrev([],
            get_nil               A2                 %      [])
            proceed                                  % .

L458:       trust_me_else         fail,2             %
L464:       allocate                                 %
            get_list              A1                 % nrev([
            unify_variable        Y1                 %      H |
            unify_variable        A1                 %      T],
            get_variable          Y2, A2             %      L) :-
            put_variable          Y3, A2             % nrev(T,RT)
            call                  nrev/2,3           % ,
            put_unsafe_value      Y3, A1             % conc(RT,
            put_list              A2                 %      [
            unify_value           Y1                 %      H |
            unify_nil                                %      []],
            put_value             Y2, A3             %      L)
            deallocate                               %
            execute               conc/3             % .
```

Performing the *transformation* on the first clause gives:

```
            get_structure         nrev/2,A1
            unify_variable        A1
            unify_variable        A2
            get_nil               A1                 % nrev([],
            get_nil               A2                 %      [])
            proceed                                  % .
```

The instructions of the code new to the clause are italicized. The switch and try_me_else instructions were not included because they form part of the *procedure* nrev/2 rather than the first clause of nrev/2. The second clause for the nrev procedure transforms to:

```
            get_structure         ':-'/2,A1
            unify_variable        A4
            unify_variable        A5
            get_structure         nrev/2,A4
            unify_variable        A1
            unify_variable        A2
            allocate                                 %
            get_list              A1                 % nrev([
            unify_variable        Y1                 %      H |
            unify_variable        A1                 %      T],
            get_variable          Y2, A2             %      L) :-
            put_variable          Y3, A2             % nrev(T,RT)
            get_structure         ','/2,A5
            unify_variable        A4
            unify_variable        A5
```

```
get_structure       nrev/2,A4
unify_local_value   A1
unify_local_value   A2
put_unsafe_value    Y3, A1        % conc(RT,
put_list            A2            %    [
unify_value         Y1            %      H |
unify_nil                         %      []],
put_value           Y2, A3        %    L)
deallocate                        %
get_structure       conc/3,A5
unify_local_value   A1
unify_local_value   A2
unify_local_value   A3
proceed
```

The reader should think of each of these transformations above as a unary unit clause. A1 should point to the structure to be matched or a variable. In the latter case, when the proceed instruction is reached, the variable will be instantiated to the clause structure. Of course, the original variable names will be missing; these may be filled in if desired by a number of different methods†.

In a real implementation, it will be undesirable to perform the actual translation. What should be done instead is to run the code for the prologue elsewhere, jump to the start of the clause code and interpret the call and execute instructions in a different manner. This interpretation of the call and execute instructions can be realized in at least two ways.

The first way of reinterpreting the call and execute instructions is to add another mode bit to the machine. This may in fact be worthwhile since *clause* and *retract* are, in some programs, quite heavily used. In one mode, the call and execute instructions behave normally; in the other mode, the sequence of *get* and *unify* instructions is performed. To set the mode bit, it may be desirable to create a new instruction which will also perform code for the prologue and branch to the start of the clause.

The second method involves setting breakpoints on the call and execute instructions. The break routine is responsible for performing the sequence of *get* and *unify* instructions corresponding to the call or execute instruction and for setting the next breakpoint if any. The very first breakpoint is set by the code that does the prologue. This second method is quite appropriate for a software implementation of the Prolog engine; implementing mode bits is quite expensive in software. On the other hand the first method, described in the preceding paragraph, may be more suitable for a hardware implementation of the Prolog engine.

In the C-based Prolog system written at Syracuse University, the second technique is used. The prologue is actually implemented as part of a builtin which returns the clause structure of a given clause.‡ This builtin also sets a breakpoint on the first call (or execute)

† If it is important to fill in the original variable names, another clause vname(DBRef,VNameList) may be asserted. DBRef is a reference to the clause in the database. VNameList is a list of the variables (as atoms) that occur in the clause from left to right with no duplications. To reinstall the variable names, the clause structure obtained from the decompilation process should be traversed from left to right. Every time a *hole* (variable) is found in the structure, it is filled in with the first element in the variable name list. After filling a hole, the first element of the list is discarded and the rest of the list is used for the remainder of the traversal.

‡ The builtin is called as clause_structure(ProcName,Arity,DBRef,Struct) where ProcName and Arity are the predicate name and arity of the clause referenced by DBRef. Struct is usually an output argument and represents the source structure of the clause. An additional builtin is provided for obtaining an initial data base reference to the first

instruction, if any, and sets the P register (program counter) to the start of the clause to decompile. When the clause is done "executing", it has decompiled itself.

Limitations

The limitations of this method have to do with the implementations of =/2, var/1, nonvar/1, and similar builtins.

Some compilers emit the following code for =/2:

```
put argument one, A1
put argument two, A2
get_value   A1,A2
```

Provided that the get_value instruction doesn't fail, the above method described will work fine; the problem is that the resulting structure won't always be the same as the original structure. The meaning of the two clauses will be the same, but syntactically, they won't be. The reason for this is that the transformation fails to take into account the fact that get_value is used in place of a call to the equality predicate. If the get_value instruction is replaced by a call to =/2 or perhaps an instruction that performs the same function as *get_value A1,A2* then the decompilation method will work.

Even worse, some compilers transform clauses with =/2 in them. For example,

```
p(X,f(g(X),Y)) :- X=h(Y).
```

may be transformed to

```
p(h(Y),f(g(h(Y)),Y)).
```

Again, the decompilation procedure will work, but probably not as expected.

Another problem results in the way that var/1 is implemented in some compilers.

```
p(c,X) :- var(X).
```

may be implemented as.

```
        get_constant        c,A1
        put_value           A2,A1
        switch_on_term      L1,fail,fail,fail
L1:     proceed
```

Again, the problem is that a transformation isn't defined for the switch_on_term instruction. It would be possible to define a transformation, but unwise since this same instruction may be used to implement nonvar also. A better approach is to create instructions which implement the var and nonvar operations and define the obvious transformations on them for the decompilation process.

Similar problems exist (in some compilers) for other operations. Either by making explicit calls or by defining new machine instructions, these problems can usually be avoided. If either making an explicit call or defining a new machine instruction is unsatisfactory or it is difficult to distinguish one abstract machine instruction from another (as in compilation to native code), a small amount of additional information may be retained in the clause header for decompilation purposes. This information is simply a vector of breakpoint offsets into the clause code. Along with the breakpoint offset may be stored the

clause of a procedure given the module, predicate name, and arity. Another builtin is used to find the next clause in the indexing scheme, failing when it finds no more. With these builtins, it is possible to implement clause/2, clause/3, and listing/1. Additional builtins designed for removing references and inserting new ones are provided for implementing retract and assert.

predicate name and arity of the call if this information is unavailable by examining the object code. It is the author's opinion that the design of the compiler and decompiler should be coupled. Correct decisions about what the compiler produces will make the process of locating calls, executes, and special code for builtins relatively painless.

Conclusions

The methods described in this paper have applications beyond implementation of clause, retract, and listing. The method of setting breakpoints may be used to implement debuggers (in particular, the standard four-port debugger). In a nutshell: breakpoints are set on the next call, execute or proceed instruction and the next failure address (obtained from the top choice point). When the breakpoint is executed, the appropriate call, redo, fail, or exit message is printed. Redo and fail messages are printed when the failure breakpoint was reached. One or more exit messages are printed when a breakpoint corresponding to a proceed instruction is executed. Call messages are printed on call and execute instructions. Because of the generalized tail recursion optimizations in the Warren architecture, it is necessary to maintain debug frames. These frames contain such information as the previous debug frame, the parent debug frame, the call structure and whether the instruction that caused this frame to be created was an execute or a call. These frames may be safely kept on the heap; in fact keeping the frames on the heap permits a clever implementation of deciding how many redo messages to print when a failure occurs.

Both the decompilation and debugging methods have been implemented in a system constructed at Syracuse University. The underlying compiler is very fast, incremental, and has an interactive interface. All of the flexibility of an interpreted system is achieved in this compiler-based system. Moreover there are no interface or portability problems as are often found in systems which require both an interpreter and a compiler.

References

Bowen, K. A. and Weinberg, T. A Meta-level Extension of Prolog. *1985 Symposium on Logic Programming*. pp. Boston, IEEE, pp. 48-53, 1985.

Clocksin, W. F. Implementation Techniques for Prolog Databases. *Software—Practice and Experience*, 15(7). pp. 669-75, 1985.

Warren D. H. D., An Abstract Prolog Instruction Set. Technical Note 309. Artificial Intelligence Center, SRI International, 1983.

Compiler Optimizations for the WAM

Andrew K. Turk

Logic Programming Research Group
School of Computer & Information Science
Syracuse University
Syracuse, NY, 13210 USA

Abstract

A series of Warren Abstract Machine (WAM) implementation techniques are presented. These techniques and compilation strategies are designed for use in a highly optimized native code Prolog compiler. A thorough knowledge of the WAM and Prolog compilation is assumed.

1. Motivations

On conventional hardware, it is necessary to compile Prolog to native machine code for optimal performance. Ideally, a native code compiler can work along side one of the many byte-code compilers which already exist. This paper outlines a series of optimizations which can be used by an intelligent compiler to translate WAM byte-codes into the native machine language of a conventional machine. Most of these strategies need not be applied to every clause and procedure; the compiler is relatively free to decide which method is best, or at least take advice from the programmer. It is assumed that the optimizations will only be applied to *static* code.

2. An Overview of Native Code

WAM byte-codes can be naively translated into native code in a straightforward fashion. Each of the steps and conditionals which must be performed by the byte-code interpreter must also be executed by native code. When these steps are directly translated into a native code stream, the overhead inherent in the interpreter is eliminated. However, much more can be done than the simple elimination of the interpreter.

3. The Read/Write Mode Bit

Warren's original design incorporates a mode bit which tells the machine whether it's in read mode or write mode. Many hardware and software interpreters actually use a mode bit. However, mode bits are expensive and difficult to manage in a native code system. Fortunately, there is a straightforward method which eliminates the bit and retains its functionality.

The mode bit is set by *get* instructions and must be maintained through the execution of the following *unify* instructions. Instructions which change the mode (either *gets* or *puts*) cannot occur between a *get* and its *unifys*. The simplest way to eliminate the mode bit is to compile each *unify* sequence twice, once for write mode and again for read mode. First, the code to dereference the register in question is emitted. Following that is a conditional branch to the read mode sequence (which directly follows the write mode sequence). Right

This work supported in part by US Air Force contract number F30602-81-C-0169.

after the conditional is the first instruction of the write mode sequence. A branch around the read mode code is placed after the last write mode instruction.

No "continuation branch" is needed after the read mode code because the next instruction corresponds to rest of the clause. *Put* instructions force the machine into write mode. This is done only so that the *unify* instructions will work properly. Since the mode is always write after a *put*, there is no need to compile a separate sequence for read mode. Some software implementations achieve the same effect with a set of *unify* instructions that always work in write mode. The compiler should emit the write mode sequence first in order to make write mode propagation more effective.

4. Write Mode propagation

Compiled constants and variables do not change the mode. However, when an unbound variable is passed into a clause at runtime, a *getStruct* or *getList* will force the machine into write mode. Not only will that particular *getStruct* or *getList* invoke write mode, but any substructure corresponding to that instruction must also run in write mode. This is because all of the *unifyVar* instructions in the original *unify* sequence run in write mode, and therefore create ʻ.esh unbound variables on the heap.

When write mode is propagated to a *getStruct F, An* or a *getList An*, the machine can infer that An contains a reference to an unbound variable on the heap. Because of this, the runtime dereference and tag check for undef can be skipped. This can amount to a significant savings for programs which spend a lot of time in head matching.

Unfortunately, there is no way to propagate read mode. Furthermore, without undue overhead, the write mode of an instruction can only be propagated to its "leftmost" subtree. This is due to the fact that there is no way to distinguish between a propagated write mode and a non-propagated write mode.

In order to implement this technique, the compiler takes advantage of the fact that the *unify* sequences are compiled once for write mode and once for read mode, with a branch in between. If the compiler detects that the mode can be propagated from some parent *get* instruction to a child *get*, the "continuation branch" in the parent's code will jump PAST the dereference and check for undef, directly into the write mode *unify* sequence of the child. If the mode for the child can be propagated, then its continuation branch will skip the dereference and check for undef of the next *get*, and so on.

Whenever the mode cannot be propagated, the parent's "continuation branch" will enter the child's *get* at the beginning. In order to capitalize on this optimization, the byte-code should match head structure in long left-descending chains so that the mode can be propagated as far as possible.

5. Shallow Backtracking on the WAM

Other Prolog implementations distinguish between two different types of backtracking: deep backtracking and shallow backtracking. Shallow backtracking occurs when the current choicepoint is the topmost object on the local stack; everything else is deep backtracking. However, we will restrict the definition of shallow backtracking to cases where some clause has failed in head matching and another clause remains to be tried in the same procedure.

Most WAM implementations have a single global subroutine or label which resets the machine and argument registers after a failure. This routine is used in the compilation of a *get* instructions when an incoming argument doesn't match. However, in order to optimize for shallow backtracking, the native code compiler should compile these actions in-line along with every *retry* and *trust* instruction.

Seen in this light, each *retry* resets the argument registers, resets the machine registers, updates a field in the choicepoint, unwinds the trail and finally jumps to the next clause. Each part of the *retry* has an entry point, and if no argument registers were changed by the previous clause, the machine can simply jump past that part of the *retry*, just as if the mode were being propagated.

Suppose that a procedure contains two clauses, numbered 0 and 1. Failure in the head of clause 0 will always cause shallow backtracking. Each instruction in the head of clause 0 which might cause failure must have some way of causing the machine to fail (i.e., branching to a failure label). Furthermore, the compiler will know how many registers might have been modified in clause 0 prior to each instruction.

Thus, the *trust* separating clause 0 from clause 1 will be arranged so that the first register modified in clause 0 is the last one reset in the *trust*. The second register modified will be the second to last reset, and so on.

By doing this, the compiler can have the first possibly failing instruction in clause 0 jump past almost all of the compiled *trust* because only a few registers have been changed. The last possibly failing instruction in clause 0 will fail into the beginning of the *trust* in order to reset all the changed registers.

Since the compiler calculates where to jump during shallow backtracking, the nextClause field in the choicepoint record is only used for deep backtracking. Clause 1 is the last clause in the example procedure, and it can only cause deep backtracking. The compiler cannot calculate the failure label in this case, and must emit code to jump to the failure label stored in the choicepoint. Failure labels in choicepoints always point at the first instruction of a compiled *retry* or *trust* in order to reset all the machine registers.

6. Improved Choicepoints

Many procedures are compiled with a type of indexing which uses two *try* instructions. These procedures are composed of a series of blocks of clauses with a *try* for each block. In addition, there is a *try* which laces all the blocks together. Under certain circumstances the machine can detect at run-time that only one clause in a particular clause can possibly match. In this case, the inner *try* instruction is not executed and the second choicepoint is not created.

Since the creation of a choicepoint is a very expensive operation in terms of both space and time, the compiler should strive to avoid unnecessary choicepoints. A closer examination of the second choicepoint occasionally used in the indexing scheme shows that all its fields except for B and the address of the next clause are duplicated from the first choicepoint. In fact, the only interesting part is the address of the next clause because the B field simply points to the first choicepoint.

Two choicepoints can be collapsed into one if the next clause field of the hypothetical second choicepoint is included in the first. In particular, the two fields are allocated together in the first choicepoint and function like a small stack. A *try* instruction pushes an address on the stack, a *retry* changes the topmost entry and a *trust* pops the stack by one.

Since present compilation technology never requires more than two choicepoints per procedure, this sub-stack can be manipulated directly as an array. A *try* simply copies the first field into the second and a *trust* copies the second into the first. Sophisticated indexing techniques which require more than two choices may become available.

In this case, it will be advantageous to have another register which points to the topmost stack entry in the most recent choicepoint.

The only restriction is that the compiler must be able to know which *try* instruction will be executed first and which *trust* will be executed last. This is because the first *try* needs to allocate the choicepoint, while the last *trust* will deallocate it.

This technique avoids a great deal of space and time overhead by collapsing multiple choicepoints into a slightly larger initial choicepoint. More importantly, it means that the compiler can calculate exactly how much local stack space will be needed by the procedure at compile time. This allows it to allocate other objects below a procedure's choicepoint, such as an environment, and to share these objects between clauses.

7. Avoiding Environment Allocation

An environment is required for any clause that has more than one subgoal. The environment must be allocated before the any use of a permanent variable and before the call to the first subgoal. Since most implementations do not have a top of stack (TOS) register, the TOS must be calculated dynamically. This involves a conditional test and possibly an indirect load and an addition. Many byte-code compilers emit code so that the allocate instructions occur as late as possible in the hopes that the clause will not match and the environment need not be allocated.

However, procedures that use at least one *try* in their indexing always calculate the TOS in order to put a choicepoint on the stack. Thus, after a choicepoint has been laid down, the B register contains the TOS. The compiler can recognize this and compile references to permanent variables as offsets from B instead of E. Once the head of the clause has successfully matched, CP and E are copied into the local stack and E is updated to reflect the new environment.

This technique cannot be applied to procedures which optionally use zero or one choicepoint. Obviously, if no choicepoint was pushed on the stack, then B does not contain the TOS. In this case, there is a good chance that the procedure will be determinate anyway, and the compiler should not be overly concerned about optimizing for failure.

8. Improved Argument Registers

There are two drawbacks in the way argument registers are reset after failure in the WAM. First, many registers will be reloaded even though they were never modified in the first place. One solution to this problem to make use of shallow backtracking. However, many registers are still reloaded from the choicepoint even though they will never be referenced due to early failure in the head. Fortunately, this problem can also be minimized by a careful compiler.

This technique involves reloading argument registers from the choicepoint of a nondeterminate procedure only when absolutely necessary. Normally, the effective address of an argument register is either a host machine register or an offset into the argument array. However, the compiler can change the effective address of the first top-level occurrence of an argument register in the code to be an offset into the current choicepoint.

Thus, the compiled *retry* instructions will not include code to restore certain argument registers. The first effective address of such a top-level argument register corresponds to the stored value of that register. The machine should not change the contents of the choicepoint, but it is free to read whatever is necessary. This amounts to treating the choicepoint as a read-only cache of saved argument registers.

Two potential difficulties arise. Some of the instructions which are optimized away in byte-code (e.g., *getVar X1, A1*) cannot be eliminated with this scheme. Since the failure code will not reload certain argument registers, the compiler must explicitly restore all such registers that appear in the clause code.

Furthermore, *trust* instructions delete the topmost choicepoint and in the process remove the initial copies of the argument registers. To eliminate this problem, the compiler must emit code which only deletes a choicepoint after the head of a clause has matched or only after the head has failed. In the latter case, the machine would delete the topmost choicepoint and then jump to the deep backtrack label of the previous choice.

A careful combination of this method and shallow backtracking should be more effective than either method alone. Register usage between clauses and the "determinateness" of the procedure will influence how the compiler emits code to reset the state after shallow backtracking.

9. Backtrackable Assignment

This next technique is not an optimization, nor is it unique to native code prolog implementations, but it was developed along with the other methods in this paper.

Backtrackable assignment appears to be a useful device for a number of different applications. It requires that extra information be stored on the trail for certain entries: those that were assigned to. The extra information is a copy of the particular cell before it was destructively assigned.

Naive implementations of backtrackable assignment either place a tag bit on each trail entry, or store a reset value with each reset address. The former requires that the tag bit be checked on each trail entry during failure and the latter doubles the size of the trail. Fortunately, a simple method exists which pays no overhead for normal trail entries.

The idea is to interleave two separate stacks, one for normal trail entries and one for reset-to-value entries. Each stack will have a pointer to the topmost element. TR points at the topmost normal trail entry, while TR' points at the topmost reset-to-value pair. This is similar to the way choicepoints and environments are interleaved on the local stack.

Each time a normal unification is trailed, an address is pushed onto the trail and TR is incremented. During a destructive assignment, a pointer to the cell, the contents of the cell, and the previous TR' are pushed on the trail with TR being incremented by. TR' is updated to reflect the new entry.

When backtracking requires cells to be reset, the machine compares the copy of TR in the backtrack frame with TR'; the higher of these two is copied into a temporary location Temp. The part of the trail between Temp and the current contents of TR represents a contiguous block of normal trail entries. The machine can loop though these, decrementing

Temp and resetting variables to undef without checking any tags or worrying about strange objects in its path. If Temp is higher than the copy of TR in the backtrack frame, then Temp points at a reset-to-value entry which resets the variable and decrements Temp accordingly. Otherwise, no reset-to-value entries occur in the current trail segment.

This process continues by repeatedly untrailing a block of normal trail entries (possibly an empty block), and then untrailing a reset-to-value entry. When all the appropriate trail entries have been removed, TR will point at the top of the trail and TR' will point at the most recent reset-to-value entry.

10. Conclusions

A number of optimized compilation techniques have been presented. An advanced compiler should be able to make use of them when it has determined that a particular procedure can be made more efficient. The optimizations are relatively independent so the compiler is not forced into a few rigid compilation models.

11. References

E. Tick
Prolog Memory-Referencing Behavior,
Research Paper 85-281, Computer Systems Laboratory,
Stanford University, 1985.

D.H.D. Warren
An Abstract Prolog Instruction Set
Technical Note 309, Artificial Intelligence Center, Computer Science
and Technology Division, SRI International, 1983.